

Manual de Angular 2



Alberto Basalo
Miguel Angel Alvarez



desarrolloweb.com/manuales/manual-angular-2.html

Introducción: Manual de Angular

Bienvenidos al Manual de Angular, que te llevará paso a paso en tu aprendizaje, conociendo las principales herramientas y patrones impulsados por el framework Javascript. Con Angular podrás desarrollar todo tipo de aplicaciones del lado del cliente, de cualquier nivel de complejidad.

En esta página irás viendo los artículos del Manual de Angular según los vayamos publicando a lo largo de los próximos meses.

Nuestro objetivo es recorrer las piezas principales de este potente framework y estudiar los mecanismos para el desarrollo de aplicaciones web universales con Angular. Encontrarás una estupenda introducción al desarrollo con el nuevo Angular, los artefactos de Angular como componentes, directivas o módulos, así como muchas prácticas adecuadas para sacar el mejor partido de esta herramienta.

También estudiaremos los servicios, cómo hacer comunicaciones HTTP y por supuesto la comunicación entre componentes de la aplicación. Aprenderás qué son los observables, cómo trabajar con el sistema de routing y mucho más.

Este manual aborda el framework Angular, en sus versiones 2 en adelante. Cabe aclarar que, poco después de salir la versión de Angular 2, el framework cambió de nombre a Angular. A partir de aquí se han ido publicando diversas entregas con números basados en el versionado semántico, pero siempre bajo la misma base tecnológica. Por tanto, este manual es válido tanto si quieres conocer Angular 2 o Angular 4... 5, 6, 7, 8 o las que vengan, ya que el framework en sí sigue manteniendo sus mismas bases descritas en estos artículos.

Encuentras este manual online en:

<http://desarrolloweb.com/manuales/manual-angular-2.html>

Autores del manual

Las siguientes personas han participado como autores escribiendo artículos de este manual.

Miguel Angel Alvarez

Fundador de DesarrolloWeb.com y la plataforma de formación online EscuelaIT. Comenzó en el mundo del desarrollo web en el año 1997, transformando su hobby en su trabajo.



Alberto Basalo

Alberto Basalo es experto en Angular y otras tecnologías basadas en Javascript, como NodeJS y MongoDB. Es director de Ágora Binaria, empresa dedicada al desarrollo de aplicaciones y a la formación a través de Academia Binaria.



Introducción al desarrollo con Angular

En esta primera parte del manual vamos a dar los primeros pasos para desarrollar aplicaciones con Angular. Veremos cómo crear el esqueleto de la aplicación básica y conoceremos los principales componentes de todo proyecto.

Introducción a Angular

Qué es Angular. Por qué se ha decidido escribir desde cero el popular framework Javascript y qué necesidades y carencias resuelve con respecto a su antecesor.

Angular 2 ha cambiado tanto que hasta el nombre es distinto. Lo conocíamos como "AngularJS" y ahora es sólo "Angular". No deja de ser una anécdota que hayan eliminado el "JS" hasta del nombre del dominio, pero es representativo. No porque ahora Angular no sea Javascript, sino porque es evolución radical.

Angular 2 es otro framework, no simplemente una nueva versión. A los que no conocían Angular 1 ésto les será indiferente, pero los que ya dominaban este framework sí deben entender que el conocimiento que necesitan adquirir es poco menos que si comenzasen desde cero. Obviamente, cuanto más experiencia en el desarrollo se tenga, más sencillo será lanzarse a usar Angular 2 porque muchas cosas sonarán de antes.

En este artículo encontrarás un poco de historia relacionada con Angular 1 y 2, junto con los motivos por los que Angular 2 es otro framework totalmente nuevo, que rompe compatibilidad hacia atrás.



Retos y necesidades de la nueva versión de Angular

Desde su creación hace ya más de 4 años, Angular ha sido el framework preferido por la mayoría de los desarrolladores Javascript. Este éxito ha provocado que los desarrolladores quieran usar el framework para más y más cosas.

De ser una plataforma para la creación de Web Apps, ha evolucionado como motor de una enorme cantidad de proyectos del ámbito empresarial y de ahí para aplicaciones en la Web

Mobile Híbrida, llevando la tecnología al límite de sus posibilidades.

Es el motivo por el que comenzaron a detectarse problemas en Angular 1, o necesidades donde no se alcanzaba una solución a la altura de lo deseable. Son las siguientes.

Javascript: Para comenzar encontramos problemas en la creación de aplicaciones debido al propio Javascript. Es un lenguaje con carácter dinámico, asíncrono y de complicada depuración. Al ser tan particular resulta difícil adaptarse a él, sobre todo para personas que están acostumbradas a manejar lenguajes más tradicionales como Java o C#, porque muchas cosas que serían básicas en esos lenguajes no funcionan igualmente en Javascript.

Desarrollo del lado del cliente: Ya sabemos que con Angular te llevas al navegador mucha programación que antes estaba del lado del servidor, comenzando por el renderizado de las vistas. Esto hace que surjan nuevos problemas y desafíos. Uno de ellos es la sobrecarga en el navegador, haciendo que algunas aplicaciones sean lentas usando Angular 1 como motor.

Por otra parte tenemos un impacto negativo en la primera visita, ya que se tiene que descargar todo el código de la aplicación (todas las páginas, todas las vistas, todas las rutas, componentes, etc), que puede llegar a tener un peso de megas.

Nota: A partir de la segunda visita no es un problema, porque ya están descargados los scripts y cacheados en el navegador, pero para un visitante ocasional sí que representa un inconveniente grande porque nota que la aplicación tarda en cargar inicialmente.

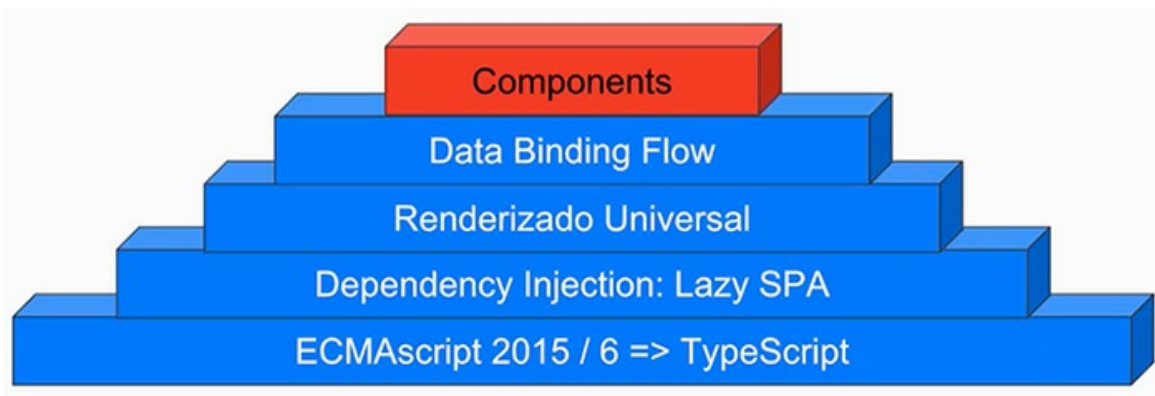
Los intentos de implementar Lazy Load, o carga perezosa, en el framework en su versión 1.x no fueron muy fructíferos. Lo ideal sería que no fuese necesario cargar toda tu aplicación desde el primer instante, pero es algo muy difícil de conseguir en la versión precedente por el propio inyector de dependencias de Angular 1.x.

Otro de los problemas tradicionales de Angular era el impacto negativo en el SEO, producido por un renderizado en el lado del cliente. El contenido se inyecta mediante Javascript y aunque se dice que Google ha empezado a tener en cuenta ese tipo de contenido, las posibilidades de posicionamiento de aplicaciones Angular 1 eran mucho menores. Nuevamente, debido a la tecnología de Angular 1, era difícil de salvar.

Soluciones implementadas en el nuevo Angular 2

Todos esos problemas, difíciles de solucionar con la tecnología usada por Angular 1, han sido los que han impulsado a sus creadores a desarrollar desde cero una nueva versión del framework. La nueva herramienta está pensada para dar cabida a todos los usos dados por los desarrolladores, llevar a Javascript a un nuevo nivel comparable a lenguajes más tradicionales, siendo además capaz de resolver de una manera adecuada las necesidades y problemas de la programación del lado del cliente.

En la siguiente imagen puedes ver algunas de las soluciones aportadas en Angular 2.



TypeScript / Javascript: Como base hemos puesto a Javascript, ya que es el inicio de los problemas de escalabilidad del código. Ayuda poco a detectar errores y además produce con facilidad situaciones poco deseables.

Nota: Con ECMAscript 6 ya mejora bastante el lenguaje, facilitando la legibilidad del código y solucionando diversos problemas, pero todavía se le exige más. Ya puestos a no usar el Javascript que entienden los navegadores (ECMAscript 5), insertando la necesidad de usar un transpilador como [Babel](#), podemos subir todavía un poco de nivel y usar TypeScript.

Angular 2 promueve el uso de TypeScript a sus desarrolladores. El propio framework está desarrollado en TypeScript, un lenguaje que agrega las posibilidades de ES6 y el futuro ES7, además de un tipado estático y ayudas durante la escritura del código, el refactoring, etc. pero sin alejarte del propio Javascript (ya que el código de Javascript es código perfectamente válido en TypeScript).

La sugerencia de usar TypeScript para desarrollar en Angular es casi una imposición porque la documentación y los generadores de código están pensados en TypeScript. Se supone que en futuro también estarán disponibles para Javascript, pero de momento no es así. De todos modos, para la tranquilidad de muchos, TypeScript no agrega más necesidad de procesamiento a las aplicaciones con Angular 2, ya que este lenguaje solamente lo utilizas en la etapa de desarrollo y todo el código que se ejecuta en el navegador es al final Javascript, ya que existe una transpilación previa.

Nota: Puedes saber más sobre este superset de Javascript en el artículo de [introducción a TypeScript](#).

Lazy SPA: Ahora el inyector de dependencias de Angular no necesita que estén en memoria todas las clases o código de todos los elementos que conforman una aplicación. En resumen, ahora con Lazy SPA el framework puede funcionar sin conocer todo el código de la aplicación, ofreciendo la posibilidad de cargar más adelante aquellas piezas que no necesitan todavía.

Renderizado Universal: Angular nació para hacer web y renderizar en HTML en el navegador,

pero ahora el renderizado universal nos permite que no solo se pueda renderizar una vista a HTML. Gracias a ésto, alguien podría programar una aplicación y que el renderizado se haga, por ejemplo, en otro lenguaje nativo para un dispositivo dado.

Otra cosa que permite el renderizado universal es que se use el motor de renderizado de Angular del lado del servidor. Es una de las novedades más interesantes, ya que ahora podrás usar el framework para renderizar vistas del lado del servidor, permitiendo un mejor potencial de posicionamiento en buscadores de los contenidos de una aplicación. Esta misma novedad también permite reducir el impacto de la primera visita, ya que podrás tener vistas "precocinadas" en el servidor, que puedes enviar directamente al cliente.

Data Binding Flow: Uno de los motivos del éxito de Angular 1 fue el data binding, pero éste tenía un coste en tiempo de procesamiento en el navegador, que si bien no penalizaba el rendimiento en todas las aplicaciones sí era un problema en aquellas más complejas. El flujo de datos ahora está mucho más controlado y el desarrollador puede direccionarlo fácilmente, permitiendo optimizar las aplicaciones. El resultado es que en Angular 2 las aplicaciones pueden llegar a ser hasta 5 veces más rápidas.

Componentes: La arquitectura de una aplicación Angular ahora se realiza mediante componentes. En este caso no se trata de una novedad de la versión 2, ya que en la versión de Angular 1.5 ya se introdujo el [desarrollo basado en componentes](#).

Sin embargo, la componetización no es algo opcional como en Angular 1.5, sino es una obligatoriedad. Los componentes son estancos, no se comunican con el padre a no ser que se haga explícitamente por medio de los mecanismos disponibles, etc. Todo esto genera aplicaciones más mantenibles, donde se encapsula mejor la funcionalidad y cuyo funcionamiento es más previsible. Ahora se evita el acceso universal a cualquier cosa desde cualquier parte del código, vía herencia o cosas como el "Root Scope", que permitía en versiones tempranas de Angular modificar cualquier cosa de la aplicación desde cualquier sitio.

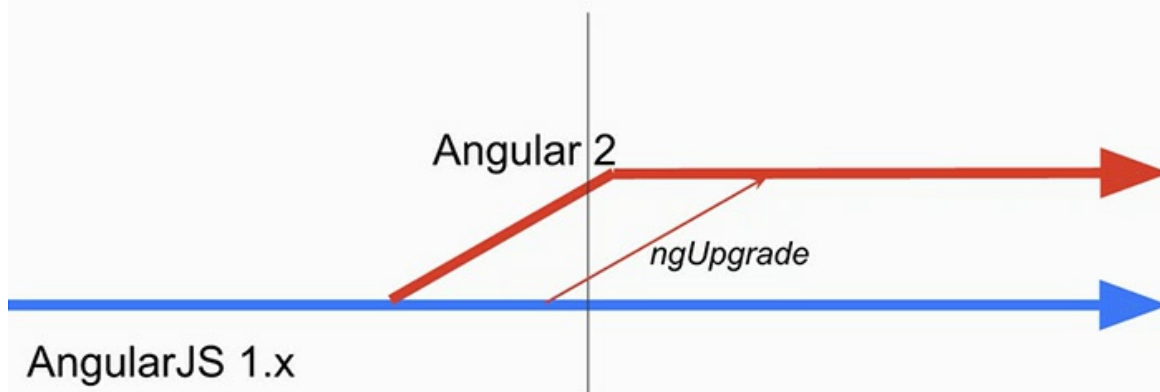
Evolución de las versiones 1 y 2 de Angular

Las versiones de AngularJS 1.x siguen vivas y continuarán dando soporte desde el equipo de Angular. Por tanto, se prevé que después de la actual 1.5 seguirán lanzando actualizaciones, nuevas versiones, etc.

La novedad es que ahora comienza en paralelo la vida de Angular 2 y seguirá evolucionando por su camino. Obviamente, la primera noticia que esperamos todos es que se presente la versión definitiva (ten en cuenta que en el momento de escribir estas líneas Angular 2 está solo en versiones RC, Release Candidate).

De este modo, debe quedar claro, para las personas que tengan aplicaciones en Angular 1.x, que no necesitan actualizarlas en una fecha determinada. Todavía existe un buen tiempo por delante en el que sus proyectos van a estar perfectamente soportados y el código de las aplicaciones perfectamente válido.

Evolución



Han anunciado además que en algún momento habrá algún sistema para hacer el upgrade de las aplicaciones de Angular 1.x a la 2.x. Esto podría permitir Incluso una convivencia, en una misma aplicación, de partes desarrolladas con Angular 1 y otras con la versión 2. La fecha de lanzamiento de ese hipotético "ngUpgrade" está sin confirmar y obviamente tampoco será magia. Veremos en el próximo próximo año lo que sucede, una vez que esa evolución de las versiones 1 y 2 estén conviviendo.

Conclusión

Se espera un futuro muy prometedor a Angular 2. Sus novedades son importantes y permitirán afrontar el futuro sobre una base tecnológica capaz de resolver todas las necesidades y retos actuales. En el Manual de Angular estaremos explicando en los próximos meses cómo usar este framework para desarrollar todo tipo de aplicaciones basadas en Javascript. Si además quieres aprender ya mismo Angular 2, tutorizado por nosotros te recomendamos también el [curso completo de Angular 2 que encuentras en EscuelaIT](#).

Para aquel desarrollador que empieza desde cero en Angular 2 será un bonito camino que le permitirá crecer profesionalmente y ampliar seriamente sus capacidades y el rango de proyectos que sea capaz de acometer. El recorrido puede ser difícil al principio, pero la recompensa será grande.

Por su parte, quien ya tenga experiencia en Angular 1.x siempre le será positiva, sobre todo para los que (a partir de la 1.5) comenzaron a usar componentes. Aunque en Angular 2 cambie la manera de realizar las cosas, le resultará todo más familiar y por tanto su curva de aprendizaje será más sencilla.

Este artículo es obra de *Alberto Basalo*
Fue publicado / actualizado en 09/06/2016
Disponible online en <https://desarrolloweb.com/articulos/introduccion-angular2.html>

Angular CLI

En este artículo te explicamos Angular CLI, el intérprete de línea de comandos de Angular permite iniciar proyectos y la creación del esqueleto (scaffolding) de todo tipo de artefactos necesarios para el desarrollo de aplicaciones.



Después de la [introducción a las características de Angular 2](#) del pasado artículo, en esta ocasión te presentamos una introducción Angular CLI, una de las herramientas esenciales para desarrollar con el nuevo framework Angular 2. Es un paso esencial y necesario antes de comenzar a ver código, puesto que necesitamos esta herramienta para poder iniciar nuestra primera aplicación Angular.

Debido a la complejidad de Angular 2, aunque el ejemplo que deseemos desarrollar se trate de un sencillo "Hola mundo", comenzar usando Angular CLI nos ahorrará escribir mucho código y nos permitirá partir de un esquema de aplicación avanzado y capaz de facilitar los flujos de desarrollo, depuración, testing o deploy. Así que vamos con ello.

Nota: Aunque nos refiramos constantemente a Angular como Angular 2, en realidad estamos explicando el CLI tal como se usa en la actualidad. Es solo porque el texto de base se publicó hace tiempo, aunque hemos ido actualizando el artículo y el manual en general para adaptarlo a los cambios del framework.

Qué es Angular CLI

Dentro del ecosistema de Angular encontramos una herramienta fundamental llamada "Angular CLI" (Command Line Interface). Es un producto que en el momento de escribir este artículo todavía se encuentra en fase beta, pero que ya resulta fundamental para el trabajo con el framework.

Angular CLI no es una herramienta de terceros, sino que nos la ofrece el propio equipo de Angular. En resumen, nos facilita mucho el proceso de inicio de cualquier aplicación con Angular, ya que en pocos minutos te ofrece el esqueleto de archivos y carpetas que vas a necesitar, junto con una cantidad de herramientas ya configuradas. Además, durante la etapa de desarrollo nos ofrecerá muchas ayudas, generando el "scaffolding" de muchos de los componentes de una aplicación. Durante la etapa de producción o testing también nos ayudará, permitiendo preparar los archivos que deben ser subidos al servidor, transpilar las fuentes, etc.

Node y npm

Angular CLI es una herramienta NodeJS, es decir, para poder instalarla necesitaremos contar con NodeJS instalado en nuestro sistema operativo, algo que podemos conseguir muy fácilmente yendo a la página de <https://nodejs.org> y descargando el instalador para nuestro sistema.

Además se instala vía "npm". Por npm generalmente no te tienes que preocupar, pues se instala al instalar NodeJS. No obstante es importante que ambas versiones, tanto la de la plataforma Node como el gestor de paquetes npm, se encuentren convenientemente actualizados. En estos momentos como requisito nos piden tener Node 4 o superior.

Nota: Puedes saber la versión de Node que tienes instalada, así como la versión de npm por medio de los comandos:

```
node -v
```

```
npm -v
```

No tienes que saber Node para desarrollar con Angular, pero sí necesitas tenerlo para poder instalar y usar Angular CLI, además de una serie de herramientas fantásticas para desarrolladores.

Instalar Angular CLI

Esto lo conseguimos desde el terminal, lanzando el comando:

```
npm install -g @angular/cli
```

Durante el proceso de instalación se instalará el propio Angular CLI junto con todas sus dependencias. La instalación puede tardar varios minutos dependiendo de la velocidad de tu conexión a Internet.

Una vez instalado dispondrás del comando "ng" a partir del cual lanzarás cualquiera de las acciones que se pueden hacer mediante la interfaz de comandos de Angular. Puedes comenzar lanzando el comando de ayuda:

```
ng --help
```

Nota: ng (que se lee "enyi") es el apelativo familiar de "Angular" que se conoce desde el inicio del framework.

También encontrarás una excelente ayuda si entras en la [página de Angular CLI](#), navegando por sus secciones, o bien en el propio [repositorio de GitHub angular-cli](#).

Crear el esqueleto de una aplicación Angular 2

Uno de los comandos que puedes lanzar con Angular CLI es el de creación de un nuevo proyecto Angular 2. Este comando se ejecuta mediante "new", seguido del nombre del proyecto que queramos crear.

```
ng new mi-nuevo-proyecto-angular2
```

Lanzado este comando se creará una carpeta igual que el nombre del proyecto indicado y dentro de ella se generarán una serie de subcarpetas y archivos que quizás por su número despisten a un desarrollador que se inicia en Angular 2. Si es así no te preocupes porque poco a poco nos iremos familiarizando con el código generado.

Además, como hemos dicho, se instalarán y se configurarán en el proyecto una gran cantidad de herramientas útiles para la etapa del desarrollo front-end. De hecho, gran cantidad de los directorios y archivos generados al crear un nuevo proyecto son necesarios para que estas herramientas funcionen. Entre otras cosas tendremos:

- Un servidor para servir el proyecto por HTTP
- Un sistema de live-reload, para que cuando cambiamos archivos de la aplicación se refresque el navegador
- Herramientas para testing
- Herramientas para despliegue del proyecto
- Etc.

Una vez creado el proyecto inicial podemos entrar en la carpeta con el comando cd.

```
cd mi-nuevo-proyecto-angular2
```

Una vez dentro de esa carpeta encontrarás un listado de archivos y carpetas similar a este:

▲ TEST-ANGULAR2

- config
- e2e
- node_modules
- public
- src
- typings
- .clang-format
- .editorconfig
- .gitignore
- angular-cli.json
- angular-cli-build.js
- package.json
- tslint.json
- typings.json

Servir el proyecto desde un web server

Angular CLI lleva integrado un servidor web, lo que quiere decir que podemos visualizar y usar el proyecto sin necesidad de cualquier otro software. Para servir la aplicación lanzamos el comando "serve".

```
ng serve
```

Eso lanzará el servidor web y lo pondrá en marcha. Además, en el terminal verás como salida del comando la ruta donde el servidor está funcionando. Generalmente será algo como esto (pero te sugerimos verificar el puerto en la salida de tu terminal):

```
http://localhost:4200/
```

En la siguiente imagen ves la salida del terminal nuestro.

```
mcMiguel:test-angular2 midesweb$ ng serve
Could not start watchman; falling back to NodeWatcher for file system events.
Visit http://ember-cli.com/user-guide/#watchman for more info.
Livereoad server on http://localhost:49152
Serving on http://localhost:4200/
```

Build successful - 831ms.

Slowest Trees	Total
-----+-----	
BroccoliTypeScriptCompiler	405ms
vendor	337ms
Slowest Trees (cumulative)	Total (avg)
-----+-----	
BroccoliTypeScriptCompiler (1)	405ms
vendor (1)	337ms

Podrías modificar el puerto perfectamente si lo deseas, simplemente indicando el puerto deseado con la opción `--port`:

```
ng serve --port 4201
```

Problema Angular CLI con Broccoli en Windows

El problema más típico que nos podemos encontrar al usar Angular CLI es que no tengamos permisos de administrador. Al intentar poner en marcha el servidor recibirás un error como este:

The Broccoli Plugin: [BroccoliTypeScriptCompiler] failed with: operation not permitted.

Es muy sencillo de solucionar en Windows, ya que simplemente necesitamos abrir el terminal en modo administrador (botón derecho sobre el icono del programa que uses para línea de comandos y "abrir como administrador". Eso permitirá que Angular CLI disponga de los permisos necesarios para realizar las tareas que requiere.

ACTUALIZACIÓN: Esto lo han cambiado en una versión beta de Angular CLI (Beta 6), por lo que ya no hace falta privilegios de administrador para usar las herramientas de línea de comandos de Angular 2.

En Linux o Mac, si te ocurre algo similar, simplemente tendrías que lanzar los comandos como "sudo". Aunque, a decir verdad, lo más conveniente es no tener que usar "sudo" para este tipo de operativas. Una solución que suele ir bien en Linux o Mac es instalar Node a través de nvm, que te ahorra problemas del uso de "sudo".

Angular CLI tiene mucho más

En esta pequeña introducción solo te hemos explicado cómo iniciar un nuevo proyecto de Angular 2 y cómo servirlo después por medio del comando `serve`. Pero lo cierto es que detrás

de Angular CLI hay muchas otras instrucciones de gran utilidad. Principalmente, como hemos comentado, encontrarás una gran cantidad de comandos que permiten crear el esqueleto de componentes, directivas, servicios, etc.

A medida que vayamos adentrándonos en el desarrollo con Angular 2 iremos aprendiendo de una forma sencilla y práctica todas las posibilidades que permite esta herramienta. De momento te recomendamos documentarte en el mencionado repositorio de Github.

Para seguir aprendiendo sobre Angular en el próximo artículo te explicaremos la [arquitectura de carpetas de una aplicación Angular](#).

Este artículo es obra de *Alberto Basalo*
Fue publicado / actualizado en 12/05/2020
Disponible online en <https://desarrolloweb.com/articulos/angular-cli.html>

Análisis de las carpetas de un proyecto básico con Angular

Angular 2 exige un marco de trabajo más concreto y avanzado. Vemos los archivos y carpetas que nos hacen falta para comenzar un proyecto básico.

En el pasado artículo abordamos la herramienta [Angular CLI](#) y ahora vamos a introducirnos en el resultado que obtenemos mediante el comando "ng new", que vimos servía para generar el esqueleto básico de una aplicación Angular 2.

No obstante, antes de entrar en materia, vamos a traer dos puntos interesantes. Uno de ellos es una reflexión previa sobre la complejidad de un proyecto "vacío" para una aplicación Angular 2. El otro es una de las preguntas típicas que se hacen cuando una persona se inicia con cualquier herramienta nueva: el editor que se recomienda usar.



Profesionalización

Todas las mejoras que nos ofrece Angular 2 tienen un coste a nivel técnico. Anteriormente (versiones 1.x) podíamos comenzar una aplicación de Angular con un código de inicio muy sencillo y sin necesidad de configurar diversas herramientas frontend. Básicamente podíamos enlazar Angular desde su CDN y con un script muy elemental empezar a usarlo.

Esto ya no es así. Con Angular 2 el número de piezas que necesitamos integrar, incluso en

aplicaciones tan sencillas como un "hola mundo", es mucho mayor.

En aplicaciones grandes, donde existía un entorno bien definido por parte del equipo de desarrollo de esa aplicación (linter, transpiler, loader, tester, deployer...) se equilibra. Es decir, Angular no nos exige nada nuevo que ya no estuvieran usando equipos de desarrollo de aplicaciones grandes. Pero lo cierto es que aplicaciones pequeñas tendrán bastante más complejidad. Todo esto nos lleva a que desarrolladores ocasionales encontrarán más difícil el uso del framework en sus pasos iniciales, pero el esfuerzo sin duda merecerá la pena porque los pondrá a un nivel muy superior y eso redundará en su beneficio profesional.

¿Qué editor de código usar con Angular 2?

Se puede usar cualquier editor de código. Es una aplicación HTML y Javascript / TypeScript, por lo que puedes usar cualquier editor de los que vienes usando para cualquiera de estos lenguajes.

Como recomendación se sugiere usar un editor ligero, pero que te facilite la programación. Entre los que encontramos de código libre y gratuitos para cualquier uso están Brackets, Atom o Visual Studio Code. Éste último es quizás el más indicado, porque ya viene configurado con una serie de herramientas útiles y clave para desarrollar con Angular 2 como es el "intellisense" de TypeScript. De todos modos, a través de plugins podrás hacer que tu editor preferido también sea capaz de mostrarte las ayudas en tiempo de programación del compilador de TypeScript.

Archivos y carpetas del proyecto

Impacta un poco que, recién creado un proyecto para Angular 2 por medio de Angular CLI, veamos en la carpeta de archivos varias subcarpetas, y varias de ellas con el contenido de cientos de ficheros. No obstante, no todo es código de tu aplicación, sino muchas veces son carpetas para crear la infraestructura de todo el tooling NodeJS incluido para gestionar una aplicación Angular 2.

Ahora conoceremos las partes que nos hacen falta para comenzar, aunque sin entrar en demasiados detalles.

Todos los archivos del raíz: Seguro que muchos de los lectores reconocen muchos de los archivos que hay dentro, como package.json (descriptor de dependencias npm) o .gitignore (archivos y carpetas que git debería ignorar de este proyecto cuando se añada al repositorio). En resumen, todo lo que encontraremos en esta raíz no son más que archivos que definen nuestro proyecto y configuran el entorno para diversas herramientas.

Nota: Observarás que no hay un index.html, porque esta no es la carpeta raíz de los archivos que se deben servir por el servidor web.

src: Es la carpeta más interesante para ti como desarrollador, ya que es el lugar donde colocarás el código fuente de tu proyecto. En realidad más en concreto la carpeta "app" que

encontrarás dentro de "src" es donde tienes que programar tu aplicación. Observarás que ya viene con diversos contenidos, entre otras cosas el index.html que debe servir como página de inicio. No obstante, no es exactamente el directorio raíz de publicación, porque al desplegar el proyecto los resultados de compilar todos los archivos se llevarán a la carpeta "dist".

En la carpeta src es donde vas a realizar todo tu trabajo como desarrollador. Seguramente otros muchos archivos te resulten familiares, como el favicon.ico.

Verás además varios archivos .ts, que son código fuente TypeScript. Como quizás sepas, los archivos .ts solo existen en la etapa de desarrollo, es decir, en el proyecto que el navegador debe consumir no encontrarás archivos .ts, básicamente porque el navegador no entiende TypeScript. Esos archivos son los que se compilarán para producir el código .js que sí entienda el navegador.

Nota: Si todavía te encuentras reticente al uso de TypeScript no te preocupes, ya que cualquier código Javascript que pongas en ese fichero es código TypeScript válido. Por tanto tú podrías perfectamente escribir cualquier código Javascript dentro de los archivos .ts y todo irá perfectamente. Si además conoces algo de TypeScript y lo quieres usar para facilitarte la vida en tiempo de desarrollo, tanto mejor para ti.

dist: Es la versión de tu aplicación que subirás al servidor web para hacer público el proyecto. En dist aparecerán todos los archivos que el navegador va a necesitar y nunca código fuente en lenguajes no interpretables por él. (Observa que no hay archivos .ts dentro de dist). Ojo, pues muy probablemente tengas que iniciar el servidor web integrado en Angular CLI para que aparezca la carpeta "dist" en el directorio de tu proyecto. Puedes obtener más información sobre cómo lanzar el servidor web en el [artículo de Angular CLI](#).

Public: Es donde colocas los archivos estáticos del proyecto, imágenes y cosas similares que se conocen habitualmente como "assets". Estos archivos también se moverán a "dist" para que estén disponibles en la aplicación una vez subida al servidor web desde donde se va a acceder.

e2e: Es para el desarrollo de las pruebas. Viene de "end to end" testing.

node_modules: Son los archivos de las dependencias que mantenemos vía npm. Por tanto, todas las librerías que se declaren como dependencias en el archivo package.json deben estar descargados en esta carpeta node_modules. Esta carpeta podría haber estado dentro de src, pero está colgando de la raíz porque vale tanto para las pruebas, como para la aplicación cuando la estás desarrollando.

tmp: Es una carpeta que no tocaremos, con archivos temporales que generará Angular CLI cuando esté haciendo cosas.

Typings: Esto son definiciones de tipos usados por las librerías que usa un proyecto en Angular 2. Estos tipos te sirven para que el editor, gracias a TypeScript, pueda informarte con el "intellisense" en el acto de escribir código, sobre las cosas relacionadas con esas librerías.

De momento eso es todo, esperamos que esta vista de pájaro te sirva de utilidad para reconocer

la estructura básica de un proyecto a desarrollar con Angular 2. En el siguiente artículo entraremos en detalle ya sobre el código, analizando por dentro algunas de estas carpetas y archivos generados desde Angular CLI y realizando nuestras primeras pruebas.

Este artículo es obra de *Alberto Basalo*
Fue publicado / actualizado en 20/06/2016
Disponible online en <https://desarrolloweb.com/articulos/analisis-carpetas-proyecto-angular2.html>

Vista de pájaro al código de una aplicación Angular (4)

Una aproximación a los elementos principales que encontramos en una aplicación Angular, versión 4, entendiendo el código.

En el capítulo anterior ofrecimos una descripción general de la [estructura de carpetas de una aplicación Angular](#), junto con algunos de sus principales archivos. Ahora vamos a introducirnos en el código y tratar de entender cómo está construida, al menos cuál es el flujo de ejecución de los archivos que vamos a encontrar.

La aplicación que vamos a radiografiar es la que conseguimos mediante el comando "ng new" del CLI. Esperamos que ya tengas tu aplicación generada. Si no es así, te recomendamos antes la lectura del artículo de [Angular CLI](#). Vamos a usar la versión 4 de Angular, aunque serviría para cualquier versión del framework a partir de la 2.

Obviamente no podremos cubrir todos los detalles, pero esperamos que de este artículo te ofrezca bastante luz sobre cómo se ejecutan los archivos principales de Angular. Verás que iremos saltando de un archivo a otro, dentro de la aplicación básica generada con "ng new", recorriendo básicamente el flujo de ejecución. Al final tendrás un resumen completo de los pasos de este recorrido que esperamos te sirva para apreciarlo de manera general.



Nota: lo cierto es que este artículo cubre Angular desde que se pasó a Webpack. En las versiones tempranas de Angular 2, usaban SystemJS para la gestión y carga de dependencias. El código de una aplicación usando esa versión antigua ya se explicó en el artículo [Zambullida en el código del proyecto inicial de Angular 2](#).

Archivo Index.html

Es de sobra sabido que las aplicaciones web comienzan por un archivo llamado index.html. Con Angular además, dado que se construyen páginas [SPA \(Single Page Application\)](#) es todavía más importante el index.html, pues en principio es el archivo que sirve para mostrar cualquier ruta de la aplicación.

El index.html en nuestra aplicación Angular, como cualquier otro archivo de los que forman parte de la app, se encuentra en el directorio "src". Si lo abres podrá llamarte la atención al menos un par de cosas:

Ausencia de Javascript:

No hay ningún Javascript de ningún tipo incluido en el código. En realidad el código Javascript que se va a ejecutar para inicializar la aplicación es inyectado por Webpack y colocado en el index.html cuando la aplicación se ejecuta o cuando se lleva a producción.

El BODY prácticamente vacío:

En el cuerpo de la página no aparece ningún HTML, salvo una etiqueta "app-root". Esta es una etiqueta no estándar, que no existe en el HTML. En realidad es un componente que mantiene todo el código de la aplicación.

Nota: Deberíamos hacer un stop al análisis del código para nombrar la "Arquitectura de componentes". En Angular y en la mayoría de librerías y frameworks actuales se ha optado por crear las aplicaciones en base a componentes. Existe un componente global, que es la aplicación entera y a su vez éste se basa en otros componentes para implementar cada una de sus partes. Cada componente tiene una representación y una funcionalidad. En resumen, las aplicaciones se construyen mediante un árbol de componentes, que se apoyan unos en otros para resolver las necesidades. En el index.html encontramos lo que sería el componente raíz de este árbol. Comenzamos a ver más detalle sobre los componentes en el siguiente artículo [Introducción a los componentes en Angular 2](#).

Archivo main.ts

Como hemos visto, no existe un Javascript definido o incluido en el index.html, pero sí se agregará más adelante para que la aplicación funcione. De hecho, si pones la aplicación en marcha con "ng serve -o" se abrirá en tu navegador y, al ver el código fuente ejecutado podrás ver que sí hay código Javascript, colocado antes de cerrar el BODY del index.html.

Ese código está generado por Webpack, e inyectado al index.html una vez la página se entrega al navegador. El script Javascript que el navegador ejecutará para poner en marcha la aplicación comienza por el archivo main.ts, que está en la carpeta "src".

Nota: Podría llamarte la atención que no es un archivo Javascript el que contiene el código

de la aplicación, pero ya dijimos en la [Introducción a Angular](#), que este framework está escrito usando el lenguaje [TypeScript](#). En algún momento ese código se traducirá, lógicamente, para que se convierta en Javascript entendible por todos los navegadores. Ese proceso lo realiza Webpack pero realmente no debe preocuparnos mucho ya que es el propio Angular CLI que va a realizar todas las tareas vinculadas con Webpack y el compilador de Javascript para la traducción del código, de manera transparente para el desarrollador.

Si abres el main.ts observarás que comienza realizando una serie de "import", para cargar distintas piezas de código. Esos import son típicos de ES6, aunque en este caso te los ofrece el propio TypeScript, ya que el transpilador de este lenguaje es el que los convertirá a un Javascript entendible por todos los navegadores.

Los import que encuentras los hay de dos tipos.

1.- imports de dependencias a librerías externas

Estos import son código de otros proyectos, dependencias de nuestra aplicación. Están gestionados por npm y han sido declarados como dependencias en el archivo package.json que deberías conocer. Tienen la forma:

```
import { enableProdMode } from '@angular/core';
```

En este caso nos fijamos en '@angular/core' que es una de las librerías dependientes de nuestra aplicación, instaladas vía npm, cuyo código está en la carpeta "node_modules". En esa carpeta, de código de terceros, nunca vamos a tocar nada.

2.- Imports a código de nuestro propio proyecto

También encontraremos imports a elementos que forman parte del propio código de la aplicación y que por tanto sí podríamos tocar. Los distingues porque tienen esta forma:

```
import { AppModule } from './app/app.module';
```

Aquí nos fijamos en la ruta del módulo que se está importando './app/app.module', que es relativa al propio main.ts.

Nota: en la ruta al archivo que se está importando falta el ".ts", ya que el archivo es un TypeScript. Pero realmente no se debe colocar, puesto que ese archivo .ts en algún momento se traducirá por un archivo .js. Para no liarse, recuerda que los import a archivos TypeScript no se coloca la extensión.

Obviamente, para profundizar habría que entender qué es cada import de los que se van realizando, los cuales tendremos tiempo de analizar llegado el momento.

Además de los imports, que es código que se va requiriendo, hay un detalle que encontramos al final del fichero:

```
platformBrowserDynamic().bootstrapModule(AppModule)
    .catch(err => console.log(err));
```

Esto es el bootstrap, el arranque, de la aplicación, que permite decirle a Angular qué tiene que hacer para comenzar a dar vida a la aplicación. Al invocar al sistema de arranque estamos indicando qué módulo (AppModule) es el principal de la aplicación y el que tiene los componentes necesarios para arrancar.

Al final, el bootstrap provocará que Angular lea el código del módulo y sepa qué componentes existen en el index, para ponerlos en marcha, provocando que la aplicación empiece a funcionar, tal como se haya programado.

Nota: fíjate que para arrancar la aplicación se hace uso de los elementos que hemos importado en las líneas anteriores, entre ellos platformBrowserDynamic y AppModule.

Archivo app.module.ts

De entre todos los imports que se hace en el main.ts hay uno fundamental, que nos sirve para tirar del hilo y ver qué es lo que está pasando por abajo, para hacer posible que todo comience a funcionar. Se trata de app.module.ts, que podemos considerar como el módulo principal de la aplicación.

En este fichero nos encontramos, como viene ya siendo habitual, varios imports de elementos que vienen de otros módulos, pero hay además dos cosas que pueden ser clave para entender el flujo de ejecución.

Import de nuestro componente raíz

En el módulo principal, app.module.ts, encontramos el import al componente raíz de la aplicación (esa etiqueta HTML no-estándar que aparecía en el BODY del index).

```
import { AppComponent } from './app.component';
```

Este componente es importante en este punto porque es el primero que estamos usando y porque es el único que te ofrecen ya construido en la aplicación básica creada con el CLI de Angular.

Al importar el componente lo que estamos obteniendo es una referencia "AppComponent", donde estará la clase creada para implementar el componente.

Decorador @NgModule

Esta es la primera vez que quizás estás conociendo los decoradores, algo que viene directamente otorgado por TypeScript. Los decoradores permiten asignar metadata a funciones, clases u otras cosas. Las funciones decoradoras tienen un nombre y las usamos para asignar esos datos, que podrían modificar el comportamiento de aquello que se está decorando.

El decorador completo en la versión de Angular que estamos usando (4) es este:

```
@NgModule({
  declarations: [
    AppComponent
  ],
  imports: [
    BrowserModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})
```

Como ves, a la función decoradora la alimentamos con un objeto pasado por parámetro, en el que indicamos diversos datos útiles para la clase AppModule. Este decorador se irá editando y agregando nuevas cosas a medida que vayamos desarrollando, por lo que nos resultará bastante familiar a poco que comencemos a desarrollar con Angular.

En el decorador hay una propiedad llamada "bootstrap", que contiene un array con los componentes que Angular tiene que dar vida, para conseguir que las cosas comiencen a funcionar: bootstrap: [AppComponent]. Esto le dice a Angular que hay en el index.html hay un componente implementado con la clase AppComponent, que tiene que ejecutar.

Componente raíz de la aplicación

Para llegar al meollo del código, tenemos que observar, aunque todavía por encima, el componente raíz. Lo habíamos usado en el index.html:

```
<app-root></app-root>
```

Pero echemos un vistazo al código del componente. Su ruta la puedes deducir del import que se ha realizado en el app.module.ts, o sea "src/app/app.component.ts".

De momento solo queremos que encuentres tu segundo decorador, con el código siguiente.

```
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
```

- El selector o etiqueta que implementa este componente: "app-root". Justamente es la etiqueta extraña que había en el index.html.
- El archivo .html su template: "./app.component.html".

- Los estilos CSS que se usan en el componente: `"/app.component.css"`

Con lo que acabas de ver, será fácil abrir encontrar el archivo donde está el template. ¿no? Puedes abrirlo y verás el código HTML que aparecía al poner en marcha la aplicación. (Comando `"ng serve -o"` del CLI)

Resumen del flujo de ejecución de la aplicación básica Angular (4)

Somos conscientes que hemos aportado muchos datos en este artículo, quizás demasiados para asimilarlos en una primera lectura. Si todo es nuevo para ti debe de ser un poco complicado hacerse un esquema perfecto del funcionamiento de las cosas, así que vamos a parar un momento para repasar de nuevo el flujo de ejecución del código.

1. En el `index.html` tenemos un componente raíz de la aplicación `"app-root"`.
2. Al servir la aplicación (`ng serve`), o al llevarla a producción (`ng build`) la herramienta Webpack genera los paquetes (bundles) de código del proyecto y coloca los correspondientes scripts en el `index.html`, para que todo funcione.
3. El archivo por el que Webpack comienza a producir los bundles es el `main.ts`
4. Dentro del `main.ts` encontramos un import del módulo principal (`AppModule`) y la llamada al sistema de arranque (`bootstrapModule`) en la que pasamos por parámetro el módulo principal de la aplicación.
5. En el módulo principal se importa el componente raíz (`AppComponent`) y en el decorador `@NgModule` se indica que este componente forma parte del bootstrap.
6. El código del componente raíz tiene el selector `"app-root"`, que es la etiqueta que aparecía en el `index.html`.
7. El template del componente raíz, contiene el HTML que se visualiza al poner en marcha la aplicación en el navegador.

Conclusión

Por fin hemos llegado a seguir la ejecución de la aplicación, para encontrar su contenido. Seguro que habiendo llegado a este punto sentirás una pequeña satisfacción. Puedes aprender más de los componentes en el artículo [Introducción a los componentes en Angular 2](#).

Somos conscientes que mucho del conocimiento ha quedado en el aire y hay muchas dudas que querrás resolver, pero con lo que hemos visto hemos conseguido el objetivo planteado, disponer de una vista de pájaro del código que encuentras al iniciar tu aplicación Angular.

Puedes continuar la lectura del [Manual de Angular](#) para obtener más información sobre decoradores, componentes, y arranque de las aplicaciones.

Este artículo es obra de *Miguel Angel Alvarez*
Fue publicado / actualizado en *05/10/2017*
Disponible online en <https://desarrolloweb.com/articulos/codigo-aplicacion-angular.html>

Zambullida en el código del proyecto inicial de Angular 2

Comenzamos a analizar el código de un proyecto básico con Angular 2, en las primeras versiones, generado con Angular CLI. Prestamos al index de la aplicación y al componente principal, que podremos editar para comprobar el funcionamiento.

En este artículo vamos a analizar el código de la aplicación inicial que se instala al hacer un nuevo proyecto con Angular CLI. Ten en cuenta que en artículos anteriores del [Manual de Angular 2](#) hemos abordado ya en una primera instancia la estructura de carpetas generada por Angular CLI, además de explicar cómo crearla a través de comandos de consola.

Por tanto, nos centraremos en entender cuál es el flujo de ejecución del código, junto con los archivos que se van incluyendo y recorriendo al ejecutar la aplicación. Obviamente, habrá cosas en las que no podamos entrar todavía en muchos detalles, pero no hay que preocuparse porque serán materia de estudio en sucesivos artículos.



Archivo package.json

Vamos a comenzar por analizar este archivo, ya que muchas cosas son declaradas inicialmente en él. Al analizar este archivo encontraremos el origen de muchas de las librerías que usa Angular 2.

Como debes saber, package.json es un archivo en el que se declaran las dependencias de una aplicación, gestionadas vía npm. Su código es un JSON en el que se declaran varias cosas.

Inicialmente hay que ver la propiedad "dependencies". En ella encontrarás la librería Angular separada en varios módulos. Esta es una de las características de la nueva plataforma de desarrollo promovida por Angular 2, la modularización del código. Encontrarás que una aplicación Angular 2 necesita ya de entrada diversos módulos como son "common", "compiler", "core", etc.

Luego hay una serie de librerías de terceros, no creadas directamente por el equipo de Angular, o bien creadas por ellos mismos pero con un enfoque universal (capaz de usarse en otros tipos de proyectos). Son "es6-shim", "rxjs", etc.

Todos estos módulos antes se incluían por medio de scripts (etiqueta SCRIPT) en el HTML de la página. Pero en esta versión y gracias a Angular CLI ya viene incorporada una mejor manera de incluir módulos Javascript, por medio de "SystemJS". Observarás que el propio SystemJS está incluido como dependencias "systemjs" y con él podríamos incluir todo tipo de código, no solo JS, sino otros ficheros necesarios como CSS.

Nota: Al describir la [estructura de carpetas del proyecto Angular 2](#) ya explicamos que todas las dependencias de package.json te las instala npm en la carpeta "node_modules".

Entendiendo lo básico de SystemJS

Para comenzar por nuestro análisis del código, vamos a abrir el archivo "src/index.html".

Como dijimos, la carpeta src es donde se encuentran las fuentes de tu proyecto. En ella encontramos un index.html que es la raíz de la aplicación. Todo empieza a ejecutarse a través de este archivo.

Te llamará la atención el código siguiente:

```
{{#each scripts.polyfills}}<script src="{{.}}"></script>{{/each}}
```

Ese código hace un recorrido por una serie de librerías y las va colocando dentro de etiquetas SCRIPT para incluir su código en la aplicación. De momento lo que debes saber sobre este código es que por medio de él se cargan librerías externas que necesita Angular y que hemos visto declaradas en las "dependencies" del archivo package.json. Lo que nos interesa saber es que así se está cargando, entre otras, la librería SystemJS.

Ahora, en el final de index.html encontrarás allí un script. Cuando llegamos a este punto, SystemJS ya está cargado y en este Script se realiza una inicialización de esta librería para cargar los elementos necesarios para comenzar a trabajar:

```
<script>
  System.import('system-config.js').then(function () {
    System.import('main');
  }).catch(console.error.bind(console));
</script>
```

Encontramos el objeto "System", que es una variable global definida por la librería SystemJS. Por medio del método "import" consigue cargar módulos. Apremiarás que se está cargando inicialmente un archivo llamado "system-config.js".

Luego vemos el método then() y catch(). Estos métodos los deberías de reconocer, pues son pertenecientes a un patrón bien conocido por los desarrolladores Javascript, el de promesas. El método then() se ejecutará cuando se termine de cargar el módulo "system-config.js" y el método catch() se ejecutaría en caso que no se haya podido cargar ese archivo. Gracias a then(), después de haber cargado "system-config.js" entonces se cargará "main", que enseguida veremos qué es.

En este punto te preguntarás ¿Dónde está system-config.js?. Quizás no lo encuentres, pero veas en la misma carpeta "src" el archivo system-config.ts. Ese es un archivo TypeScript que contiene el código de system-config.js antes de transpilar con el TypeScript Compiler.

Nota: TypeScript es un lenguaje para el programador. Lo que usa el navegador es Javascript. El TypeScript compiler se encargará de hacer esa conversión del .ts a un .js.

El archivo system-config.ts generalmente no lo vas a tener que tocar, porque Angular CLI te lo irá actualizando. Si lo abres sin duda irás reconociendo algunas líneas, cosas que necesitará SystemJS. No vamos a entrar ahora en el detalle, de momento quédate con que es código generado.

Por su parte la referencia a "main" que teníamos antes en los import del index.html System.import('main'), es un import. No le ponen ni siquiera la extensión del archivo "main.js" y esto es porque "main" es un alias declarado en el system-config.ts. Fíjate en el código del archivo, en estas líneas:

```
// Apply the CLI SystemJS configuration.
System.config({
  map: {
    '@angular': 'vendor/@angular',
    'rxjs': 'vendor/rxjs',
    'main': 'main.js'
  },
  packages: cliSystemConfigPackages
});
```

El objeto "map" tiene una lista de alias y archivos que se asocian. Siendo que "main" corresponde con "main.js". Nuevamente, no encontrarás un "main.js" entre los archivos del proyecto, en la carpeta "src", porque lo que tendremos es un main.ts que luego se convertirá en el main.js.

Ahora puedes abrir main.js. Verás que su código nuevamente hace uso de SystemJS, realizando diversos imports. Estos imports son como los que conoces en ECMAScript 6 y básicamente lo que te traen son objetos de diversas librerías.

Encontrarás este código en main.js, o algo similar:

```
import { bootstrap } from '@angular/platform-browser-dynamic';
import { enableProdMode } from '@angular/core';
import { TestAngular2AppComponent, environment } from './app';
```

Por ejemplo, estás diciéndole que importe el objeto "bootstrap" de la librería "@angular/platform-browser-dynamic". Esa librería está declarada dentro de "system-config.ts"

Luego verás otras líneas, que es el bootstrap, o inicio de la aplicación Angular. No vamos a entrar en detalle, pero es lo equivalente al "ng-app" que colocabas antes en el código HTML de tu index. Esto, o algo parecido ya se podía hacer con Angular 1 y se conocía como el arranque manual de Angular.

Componente principal de una aplicación

Seguimos analizando index.html y encontramos en el código, en el cuerpo (BODY) una etiqueta que llamará la atención porque no es del HTML tradicional. Es el uso de un componente y su código será algo como:

```
<mi-nombre-proyecto-app>Loading...</mi-nombre-proyecto-app>
```

Ese es el componente raíz de nuestra aplicación Angular 2. Hablaremos de componentes con detalle más adelante. De momento para lo que te interesa a ti, que es reconocer el flujo de ejecución básico, hay que decir que su código está en la carpeta "src/app".

En esa carpeta encontrarás varios archivos del componente que analizaremos con calma más adelante. De momento verás un archivo ".html" que contiene la vista de este componente y un archivo ".css" que contiene el CSS. Si tu componente se llamaba "mi-nombre-proyecto-app", estos archivos se llamarán "mi-nombre-proyecto.component.html" y "mi-nombre-proyecto.component.css".

Para terminar esta primera zambullida al código te recomendamos editar esos archivos. Es código HTML y CSS plano, por lo que no tendrás ningún problema en colocar cualquier cosa, siempre que sea HTML y CSS correcto, claro está.

Para quien use Angular 1 ya reconocerá una estructura como esta:

```
{{title}}
```

Es una expresión que Angular 2 sustituirá por un dato que se declara en el archivo .ts del componente. De momento lo dejamos ahí.

Ejecutar el proyecto

Para comprobar si tus cambios en el HTML del componente han tenido efecto puedes probar el proyecto. La ejecución de esta aplicación ya la vimos en el artículo de [Angular CLI](#), por lo que no necesitarás mayores explicaciones. De todos modos, como resumen, sigue los pasos:

Desde la raíz del proyecto, con el terminal, ejecutamos el comando:

```
ng serve
```

Luego nos dirigimos a la URL que nos indican como resultado de la ejecución del comando, que será algo como:

```
http://localhost:4200/
```

Entonces deberías ver la página funcionando en tu navegador, con el HTML editado tal como

lo has dejado en el archivo .html del componente principal.

En futuros artículos profundizaremos sobre muchos de los puntos relatados aquí. De momento creemos que esta introducción al código debe aclararte muchas cosas, o plantearte muchas otras dudas.

Si es tu caso, no te preocupes por sentirte despistado por tantos archivos y tantas cosas nuevas, pues poco a poco iremos familiarizándonos perfectamente con toda esta infraestructura. Para tu tranquilidad, decir que esta es la parte más compleja y que, a partir de aquí, las cosas serán más agradecidas. Si además vienes de Angular 1, empezarás a reconocer mejor las piezas que antes existían en el framework.

Este artículo es obra de *Alberto Basalo*
Fue publicado / actualizado en 29/06/2016
Disponible online en <https://desarrolloweb.com/articulos/codigo-proyecto-inicial-angular2.html>

Introducción a los componentes en Angular

Un primer acercamiento al mundo de los componentes en Angular (2 en adelante), a través del componente inicial que se crea en todo nuevo proyecto. Aprenderás a reconocer sus partes y realizaremos unas modificaciones.

En Angular 2 se desarrolla en base a componentes. Desde la aplicación más básica de Angular 2, el Hola Mundo, todo tiene que comenzar por un componente. Nuestra aplicación "componetizada" constará la verdad de un árbol de componentes de n niveles, desde un componente principal a sus hijos, nietos, etc.

Por si no lo sabes, los componentes son como etiquetas HTML nuevas que podemos inventarnos para realizar las funciones que sean necesarias para nuestro negocio. Pueden ser cosas diversas, desde una sección de navegación a un formulario, o un campo de formulario. Para definir el contenido de esta nueva etiqueta, el componente, usas un poco de HTML con su CSS y por supuesto, un poco de Javascript para definir su funcionalidad.

Básicamente eso es un componente, una de las piezas fundamentales de las aplicaciones en Angular, que nos trae diversos beneficios que mejoran sensiblemente la forma de organización de una aplicación, su mantenimiento, reutilización del código, etc.



Para comenzar a introducirnos en el desarrollo en base a componentes vamos a realizar en este primer artículo un análisis del componente inicial, que contiene de toda aplicación Angular y que podemos encontrar en el proyecto básico creado vía [Angular CLI](#).

Localizar el componente inicial

En el [Manual de Angular 2](#) hemos visto que nuestra aplicación se desarrolla en el directorio "src". Allí encontramos el archivo index.html raíz de la aplicación.. Si lo abres verás que no tiene ningún contenido en sí. Apenas encontrarás el uso de un componente, una etiqueta que no pertenece al HTML y que se llama de una manera similar al nombre que le diste al proyecto, cuando lo creaste con Angular CLI.

```
<proyecto-angular2-app>Loading...</proyecto-angular2-app>
```

Este es el componente donde tu aplicación Angular 2 va a vivir. Todos los demás componentes estarán debajo de éste, unos dentro de otros. Todo lo que ocurra en tu aplicación, estará dentro de este componente.

El texto que está dentro del componente (Loading...) es lo que aparecerá en el navegador mientras no carga la página. Una vez que la aplicación se inicie, Angular lo sustituirá por el contenido que sea necesario.

Nota: Si al arrancar la aplicación (ng-serve) ves que ese mensaje de "Loading..." tarda en irse es porque estás en modo de desarrollo y antes de iniciarse la app tienen que hacerse varias tareas extra, como transpilado de código, que no se necesitarán hacer cuando esté en producción.

Entendiendo el código del componente

El código de este componente está generado de antemano en la carpeta "src/app". Allí encontrarás varios ficheros que forman el componente completo, separados por el tipo de código que colocarás en ellos.

proyecto-angular2.component.html: Equivale a lo que conocemos por "vista" en la arquitectura MVC.

`proyecto-angular2.component.css`: Permite colocar estilos al contenido, siendo que éstos están encapsulados en este componente y no salen para afuera.

`proyecto-angular2.component.ts`: Es el archivo TypeScript, que se traducirá a Javascript antes de entregarse al navegador. Sería el equivalente al controlador en el MVC.

`proyecto-angular2.component.spec.ts`: Un archivo TypeScript destinado a tareas de testing de componentes.

El archivo HTML, o lo que sería la vista, admite toda clase de código HTML, con etiquetas estándar y el uso de otros componentes. Además podemos colocar expresiones (entre dobles llaves), expresar binding entre componentes, eventos, etc.

Nota: Los que no conozcan de todo eso que estamos hablando no se preocupen, porque lo veremos más adelante.

Entre el HTML de la vista encontrarás:

```
{{title}}
```

Eso es una expresión. Angular lo sustituirá por el contenido de una variable "title" antes de mostrarlo al cliente. Esa variable se define en lo que sería el controlador en el patrón MVC, solo que en Angular 2 no se le llama controlador, o "controller". Ahora es una clase que podemos encontrar en el archivo TypeScript "proyecto-angular2.component.ts". Si lo abres encontrarás varias cosas.

- El import de "component" dentro de @angular/core
- Una función decoradora que hace la acción de registrar el componente
- La clase que hace las veces de controlador

La función decoradora observarás que declara diversas cuestiones.

```
@Component({
  moduleId: module.id,
  selector: 'proyecto-angular2-app',
  templateUrl: 'proyecto-angular2.component.html',
  styleUrls: ['proyecto-angular2.component.css']
})
```

Una de ellas es el "selector" de este componente, o el nombre de la etiqueta que se usará cuando se desee representar. Luego está asociada la vista (propiedad "templateUrl") al archivo .html del componente y su estilo (propiedad "styleUrls") a un array de todas las hojas de estilo que deseemos.

En la clase del componente, que se debe colocar con un export para que se conozca fuera de este módulo, es la parte que representa el controlador en una arquitectura MVC. En ella

colocaremos todas las propiedades y métodos que se deseen usar desde la vista.

```
export class ProyectoAngular2AppComponent {  
  title = 'proyecto-angular2 works!';  
}
```

Esas propiedades representan el modelo de datos y se podrán usar expresiones en las vistas para poder visualizarlas.

Nota: Observa además que el nombre de la clase de este componente tiene una forma especial. Mientras que el nombre de la etiqueta del componente (su "selector") tiene las palabras separadas por comas, aquí tenemos una notación "CamelCase" típica de Javascript. Esto es una constante. En el HTML que no se reconocen mayúsculas y minúsculas se separa por guiones, colocando todo en minúscula y los mismos nombres en Javascript se escriben con "CamelCase", todo junto y con la primera letra de cada palabra en mayúscula.

Alterando el código de nuestro componente

Para terminar este artículo vamos a hacer unos pequeños cambios en el código del componente para comprobar si la magia de Angular está funcionando.

Algo muy sencillo sería comenzar por crear una nueva propiedad en la clase del componente. Pero vamos además a colocar un método para poder usarlo también desde la vista.

```
export class ProyectoAngular2AppComponent {  
  title = 'Manual de Angular 2 de DesarrolloWeb.com';  
  visible = false;  
  decirAdios() {  
    this.visible = true;  
  }  
}
```

Nota: Esta clase "class" se escribe en un archivo TypeScript, pero realmente lo que vemos es todo Javascript válido en ES6. TypeScript entiende todo ES6 e incluso algunas cosas de ES7.

Ahora vamos a ver el código HTML que podría tener nuestra vista.

```
<h1>  
  {{title}}  
</h1>  
<p [hidden]="!visible">  
  Adiós  
</p>  
<button (click)="decirAdios()">Decir adiós</button>
```

En este HTML hemos incluido más cosas de las que puedes usar desde Angular. Habíamos mencionado la expresión, entre llaves. También encuentras el uso de una propiedad de un elemento, como es "hidden", entre corchetes (nuevo en Angular 2). Además de la declaración de un evento "click" que se coloca entre paréntesis.

En este caso lo que tenemos en la propiedad "visible" de la clase, que hace las veces de controlador, se usa para asignarlo a la propiedad hidden del elemento "p". El método de la clase, decirAdios() se usa para asociarlo como manejador del evento "click".

Hablaremos más adelante de todas estas cosas que puedes colocar en las vistas y algunas otras, junto con las explicaciones sobre la sintaxis que se debe usar para declararlas.

Nota: Al modificar los archivos del componente, cualquiera de ellos, tanto el html, css o ts, se debería refrescar automáticamente la página donde estás visualizando tu proyecto una vez puesto en marcha con el comando "ng serve", gracias al sistema de "live-reload" que te monta Angular CLI en cualquier proyecto Angular 2.

Otra cosa interesante del entorno de trabajo es que, si usas Visual Studio Code u otro editor con los correspondientes plugin TypeScript, te informarán de posibles errores en los archivos .js. Es una ayuda muy útil que aparece según estás escribiendo.

Con esto acabamos nuestro primer análisis y modificaciones en el componente inicial. Estamos seguros que esta última parte, en la que hemos modificado el código del componente básico, habrá resultado ya algo más entretenida.

Este artículo es obra de *Alberto Basalo*
Fue publicado / actualizado en 06/07/2016
Disponible online en <https://desarrolloweb.com/articulos/introduccion-componentes-angular2.html>

Sintaxis para las vistas en Angular

Explicaciones generales de la sintaxis en las vistas de los componentes de Angular:
Expresiones, binding a propiedades, declaración de manejadores de eventos y doble binding.



En los artículos anteriores del [Manual de Angular](#) hemos analizado la estructura de una aplicación básica. Una de las cosas fundamentales que encontramos es el componente principal, sobre el cual hemos hecho pequeñas modificaciones para comprobar que las cosas están funcionando y comenzar a apreciar el poder de Angular.

Dentro del componente básico encontramos varios archivos y uno de ellos es el código HTML del componente, al que comúnmente nos referiremos con el nombre de "vista", denominación que viene por el patrón de arquitectura MVC. Pues bien, en ese código HTML, la vista, de manera declarativa podemos definir y usar muchas de las piezas con las que contamos en una aplicación: Vistas, propiedades, eventos, bindeo...

La novedad en Angular, para los que vienen de la versión anterior del framework, es que ahora podemos ser mucho más precisos sobre cómo queremos que la información fluya entre componentes, entre la vista y el modelo, etc. En este artículo vamos a ofrecer una primera aproximación general a todo lo que se puede declarar dentro de una vista, teniendo en cuenta que muchas de las cosas necesitarán artículos específicos para abordarlas en profundidad.

Nota: En este artículo, así como a lo largo del manual, hacemos uso constantemente de conceptos del MVC. Recordamos que existe un artículo genérico de lo que es el MVC, [introducción al modelo vista controlador](#). De todos modos, en Angular 2 no se aplica una implementación perfectamente clara sobre el MVC. Existe una separación del código por responsabilidades, lo que ya nos aporta los beneficios de la arquitectura por capas, pero la implementación y características de estas capas no queda tan definida como podría darse en un framework backend. En este artículo y los siguientes verás que hacemos referencia a los modelos y realmente no es que exista una clase o algo concreto donde se coloca el código del modelo. Ese modelo realmente se genera desde el controlador y para ello el controlador puede obtener datos de diversas fuentes, como servicios web. Esos datos, tanto propiedades como métodos, se podrán usar desde la vista. Es algo más parecido a un VW (View Model), o sea, un modelo para la vista, que se crea en el controlador. Por otra parte, tampoco existe un controlador, sino una clase que implementa el View Model de cada componente. En resumen, conocer el MVC te ayudará a entender la arquitectura propuesta por Angular 2 y apreciar sus beneficios, pero debemos mantener la mente abierta para no confundirnos con conocimientos que ya podamos tener de otros frameworks.

Piezas declarables en una vista

Comenzaremos por describir las cosas que disponemos para su declaración en una vista, de modo que todos podamos usar un único vocabulario.

- **Propiedad:** Cualquier valor que podemos asignar por medio de un atributo del HTML. Ese elemento puede ser simplemente un atributo del HTML estándar, un atributo implementado mediante el propio Angular 2 o un atributo personalizado, creado para un componente en específico.
- **Expresión:** Es un volcado de cualquier información en el texto de la página, como contenido a cualquier etiqueta. La expresión es una declaración que Angular procesará y sustituirá por su valor, pudiendo realizar pequeñas operaciones.
- **Binding:** Es un enlace entre el modelo y la vista. Mediante un binding si un dato cambia en el modelo, ese cambio se representa en la vista. Pero además en Angular se introduce el "doble binding", por el cual si un valor se modifica en la vista, también viaja hacia el modelo.
- **Evento:** es un suceso que ocurre y para el cual se pueden definir manejadores, que son funciones que se ejecutarán como respuesta a ese suceso.

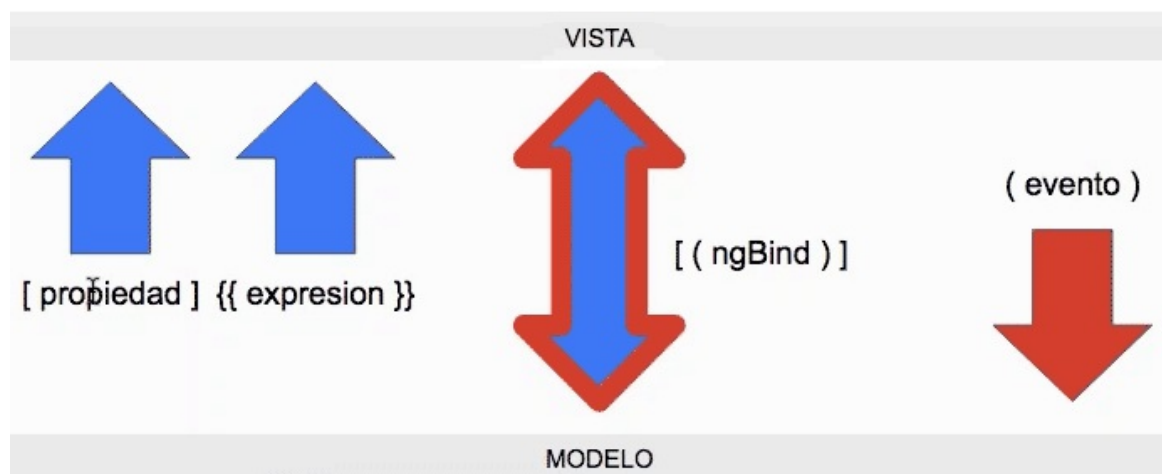
Nota: Generalmente cuando hablemos de "binding" en la mayoría de las ocasiones nos referimos a "doble binding", que es la principal novedad que trajo Angular 1 y que le produjo tanto éxito para este framework.

El problema del doble binding es que tiene un coste en tiempo de procesamiento, por lo que si la aplicación es muy compleja y se producen muchos enlaces, su velocidad puede verse afectada. Es el motivo por el que se han producido nuevas sintaxis para poder expresar bindings de varios tipos, de una y de dos direcciones. Dicho de otra manera, ahora se entrega al programador el control del flujo de la información, para que éste pueda optimizar el rendimiento de la aplicación.

Flujo de la información de la vista al modelo y modelo a vista

El programador ahora será capaz de expresar cuándo una información debe ir del modelo hacia la vista y cuándo debe ir desde la vista al modelo. Para ello usamos las anteriores "piezas" o "herramientas" en el HTML, las cuales tienen definida de antemano un sentido para el flujo de los datos.

En el siguiente diagrama puedes ver un resumen del flujo de la información disponible en Angular, junto con las piezas donde podemos encontrarlo y su sintaxis.



1. Las propiedades tienen un flujo desde el modelo a la vista. Una información disponible en el modelo se puede asignar como valor en un elemento del HTML mediante una propiedad, usando la notación corchetes. Por ej: `[propiedad]`
2. Las expresiones también viajan desde el modelo a la vista. La diferencia de las propiedades es que en este caso las usamos como contenido de un elemento y además que se expresan con dobles llaves. Por ej: `{{expresión}}`
3. El binding (a dos sentidos, o doble binding) lo expresamos entre corchetes y paréntesis. En este caso la información fluye en ambos sentidos, desde el modelo a la vista y desde la vista al modelo. Por ej: `[(ngBind)]`
4. Los eventos no es que necesariamente hagan fluir un dato, pero sí se considera un flujo de aplicación, en este caso de la vista al modelo, ya que se originan en la vista y generalmente sirven para ejecutar métodos que acabarán modificando cosas del modelo. Por ej: `(evento)`

Nota: Como ves, ahora existen diversas sintaxis para expresar cosas en las vistas. Quizás nos resulte extraño, pero enseguida nos familiarizaremos. La notación más rara es la que usamos para expresar un binding en dos direcciones `[(ngBind)]`, pero una manera sencilla de acordarnos de ella es con su denominación anglosajona "banana in a box". Los paréntesis parecen una banana, dentro de los corchetes, que parecen una caja.

Ejemplos de sintaxis utilizada en vistas de Angular 2

Realmente ya hemos visto ejemplos de buena parte de las piezas posibles a declarar en una vista. Si te fijas en el artículo anterior dedicado a la [Introducción a los componentes en Angular 2](#). Ahora les podemos dar nombres a cada uno de los elementos encontrados.

Propiedades: Era el caso de la propiedad "hidden" que usamos para mostrar / ocultar determinados elementos. En este caso, hidden no es una propiedad estándar del HTML, sino que está generada por Angular 2 y disponible para aplicar tanto en etiquetas HTML comunes como en componentes personalizados.

```
<p [hidden]="!visible">Adiós</p>
```

También podríamos aplicar valores a atributos del HTML con datos que están en propiedades del modelo, aunque no soporta todos los atributos del HTML estándar. Por ejemplo, podríamos asignar una clase CSS (class) con lo que tuviésemos en una propiedad del modelo llamada "clase".

```
<div [class]="clase">Una clase marcada por el modelo</div>
```

O el enlace de un enlace podríamos también definirlo desde una variable del modelo con algo como esto:

```
<a [href]="enlace">Pulsa aquí</a>
```

Pero en general, el uso más corriente que se dará a esta forma de definir una propiedad personalizada de un componente, de modo que podamos personalizar el estado o comportamiento del componente mediante datos que tengamos en el modelo. Veremos este caso más adelante cuando analicemos con mayor detalle los componentes.

Lo que de momento debe quedar claro es que las propiedades van desde el modelo a la vista y, por tanto, si se modifica el valor de una propiedad en el modelo, también se modificará la vista. Pero, si dentro de la vista se modifica una propiedad no viajará al modelo automáticamente, pues el enlace es de una sola dirección.

Expresiones: Es el caso de la propiedad del modelo "title" que se vuelca como contenido de la página en el encabezamiento.

```
<h1>
  {{title}}
</h1>
```

Simplemente existe esa sustitución del valor de la propiedad, colocándose en el texto de la página. El enlace es de una única dirección, desde el modelo a la vista. En la vista tampoco habría posibilidad de modificar nada, porque es un simple texto.

El caso de propiedades del HTML con valores que vienen del modelo también podríamos implementarlo por medio de expresiones, tal como sigue.

```
<a href="{{enlace}}">Clic aquí</a>
```

Nota: Las dos alternativas (usar expresiones con las llaves o los corchetes para propiedades, como hemos visto para el ejemplo de un href de un enlace cuyo valor traes del modelo) funcionan exactamente igual, no obstante para algunos casos será mejor usar la sintaxis de propiedades en vez de la de expresiones, como es el caso del enlace. Algo que entenderemos mejor cuando lleguemos al sistema de rutas.

Eventos: Esto también lo vimos en el artículo anterior, cuando asociamos un comportamiento al botón. Indicamos entre paréntesis el tipo de evento y como valor el código que se debe de ejecutar, o mejor, la función que se va a ejecutar para procesar el evento.

```
<button (click)="decirAdios()">Decir adiós</button>
```

Con respecto a Angular 1.x entenderás que ahora todas las directivas como ng-click desaparecen, dado que ahora los eventos solo los tienes que declarar con los paréntesis. Esto es interesante ya de entrada, porque nos permite definir de una única manera cualquier evento estándar del navegador, pero es más interesante todavía cuando comencemos a usar componentes personalizados, creados por nosotros, que podrán disparar también eventos personalizados. Capturar esos eventos personalizados será tan fácil como capturar los eventos estándar del HTML.

Binding: Para este último caso no hemos visto todavía una implementación de ejemplo, pero lo vamos a conseguir muy fácilmente. Como se dijo, usamos la notación "banana in a box" para conseguir este comportamiento de binding en dos direcciones. Los datos viajan de la vista al modelo y del modelo a la vista.

```
<p>  
¿Cómo te llamas? <input type="text" [(ngModel)]="quien">  
</p>
```

En este caso, desde el HTML estaríamos creando una propiedad dentro del modelo. Es decir, aunque no declaremos la propiedad "quien" en el Javascript, por el simple hecho de usarla en la estructura de binding va a producir que se declare automáticamente y se inicialice con lo que haya escrito en el campo de texto. Si la declaramos, o la inicializamos desde la clase que hace las veces de controlador, tanto mejor, pero no será necesario.

Nota: La notación "banana in a box" tiene una explicación y es que usa tanto el flujo de datos desde el modelo a la vista, que conseguimos con los corchetes para las propiedades, como el flujo desde la vista al modelo que conseguimos con los paréntesis para los eventos.

Para quien conozca Angular 1, ngModel funciona exactamente igual que la antigua directiva. En resumen, le asignamos el nombre de una propiedad en el modelo, en este caso "quien", con la que se va a conocer ese dato. A partir de entonces lo que haya escrito en el campo de texto viajará de la vista al modelo y si cambia en el modelo también se actualizará la vista, produciendo el binding en las dos direcciones.

Si quisiéramos visualizar ese dato en algún otro de la vista, por ejemplo en un párrafo, usaríamos una expresión. Por ejemplo:

```
<p>  
Hola {{quien}}  
</p>
```

Conclusión



Hemos aprendido a usar muchas de las declaraciones de Angular, que usaremos en las vistas de los componentes, que nos permiten enlazar datos y comportamientos definidos en la parte del Javascript. No te preocupes si no hemos aportado suficientes ejemplos, porque estos conocimientos los iremos aplicando constantemente a lo largo de todo el manual.

En el siguiente artículo cambiaremos de tercio, para dedicarnos a [estudiar en profundidad los componentes de Angular](#).

Este artículo es obra de *Miguel Angel Alvarez*

Fue publicado / actualizado en 17/10/2020

Disponible online en <https://desarrolloweb.com/articulos/sintaxis-vistas-angular2.html>

Los componentes en Angular

Abordamos el desarrollo basado en componentes con todo detalle. Es la pieza más importante de Angular que nos permitirá no solo estructurar una aplicación de una manera ordenada, sino encapsular funcionalidad y facilitar una arquitectura avanzada y de fácil mantenimiento de los proyectos Javascript con la nueva versión de este framework.

El concepto de los componentes en Angular y su arquitectura

Qué es un componente para Angular y cómo usaremos los componentes para realizar la arquitectura de una aplicación.

En este artículo del [Manual de Angular 2](#) queremos abordar el concepto del componente desde un punto de vista teórico, sin entrar a ver cómo se construyen en Angular 2. Esa parte práctica la dejaremos para el próximo artículo, aunque cabe recordar, para los que se impacientan por ver código, que ya dimos una [zambullida en el código de los componentes](#) cuando comenzamos con el "hola mundo".

El objetivo es entender mejor cuál es la arquitectura promovida por Angular 2 para el desarrollo de aplicaciones y qué papel específico desempeñan los componentes. Es importante porque toda aplicación Angular 2 se desarrolla en base a componentes y porque es algo relativamente nuevo en el framework.



Árbol de componentes

Una aplicación Angular 2 se desarrolla a base de crear componentes. Generalmente tendrás un árbol de componentes que forman tu aplicación y cada persona lo podrá organizar de su manera preferida. Siempre existirá un componente padre y a partir de ahí podrán colgar todas las ramas que sean necesarias para crear tu aplicación.

Esto no resultará nada extraño, pues si pensamos en una página web tenemos un mismo árbol de etiquetas, siendo BODY la raíz de la parte del contenido. La diferencia es que las etiquetas generalmente son para mostrar un contenido, mientras que los componentes no solo

encapsulan un contenido, sino también una funcionalidad.

En nuestro árbol, como posible organización, podemos tener en un primer nivel los bloques principales de la pantalla de nuestra aplicación.

- Una barra de herramientas, con interfaces para acciones principales (lo que podría ser una barra de navegación, menús, botonera, etc.).
- Una parte principal, donde se desplegarán las diferentes "pantallas" de la aplicación.
- Un área de logueo de usuarios.
- Etc.

Nota: Obviamente, ese primer nivel de componentes principales lo dictará el propio proyecto y podrá cambiar, pero lo anterior nos sirve para hacernos una idea.

Luego, cada uno de los componentes principales se podrá subdividir, si se desea, en nuevos árboles de componentes.

- En la barra de herramientas principal podríamos tener un componente por cada herramienta.
- En el área principal podríamos tener un componente para cada "pantalla" de la aplicación o "vista". A su vez, dentro de cada "vista" o "pantalla" podíamos tener otra serie de componentes que implementen diversas funcionalidades.
- Etc.

Los niveles del árbol serán los que cada aplicación mande, atendiendo a su complejidad, y cada desarrollador estime necesario, en función de su experiencia o preferencias de trabajo. A medida que componetizamos conseguimos dividir el código de la aplicación en piezas menores, con menor complejidad, lo que seguramente sea beneficioso.

Si llegamos a un extremo, y nos pasamos en nuestra ansia de componetizar, quizás obtengamos el efecto contrario. Es decir, acabemos agregando complejidad innecesaria a la aplicación, puesto que existe un coste de tiempo de trabajo y recursos de procesamiento para posibilitar el flujo de comunicación entre componentes.

Componentes Vs directivas

En Angular 2 perdura el concepto de directiva. Pero ahora tenemos componentes y la realidad es que ambos artefactos se podrían aprovechar para usos similares. La clave en este caso es que los componentes son piezas de negocio, mientras que las directivas se suelen usar para presentación y problemas estructurales.

Puedes pensar en un componente como un contenedor donde solucionas una necesidad de tu aplicación. Una interfaz para interacción, un listado de datos, un formulario, etc.

Para ser exactos, en la documentación de Angular 2 nos indican que un componente es un tipo de directiva. Existen tres tipos de directivas:

Componentes: Un componente es una directiva con un template. Habrá muchas en tu aplicación y resuelven necesidades del negocio. **Directivas de atributos:** Cambian la apariencia o comportamiento de un element. Por ejemplo tenemos `ngClass`, que nos permite colocar una o más clases de CSS (atributo `class`) en un elemento. **Directivas estructurales:** Son las que realizan cambios en el DOM del documento, añadiendo, manipulando o quitando elementos. Por ejemplo `ngFor`, que nos sirve para hacer una repetición (similar al `ngRepeat` de Angular 1.x), o `ngIf` que añade o remueve elementos del DOM con respecto a una expresión condicional.

Por tanto, a diferencia de otras librerías como Polymer, donde todo se resuelve mediante componentes, hay que tener en cuenta qué casos de uso son los adecuados para resolver con un componente.

Las partes de un componente

Aunque también hemos analizado anteriormente, cuando [repasamos la aplicación básica de Angular 2](#) generada con [Angular CLI](#), cuáles son las partes fundamentales de un componente, vamos a volver a este punto para poder ver sus piezas de manera global.

Un componente está compuesto por tres partes fundamentales:

- Un template
- Una clase
- Una función decoradora

Las dos primeras partes corresponden con capas de lo que conocemos como MVC. El template será lo que se conoce como vista y se escribe en HTML y lo que correspondería con el controlador se escribe en Javascript por medio de una clase (de programación orientada a objetos).

Por su parte, tenemos el decorador, que es una especie de registro del componente y que hace de "pegamento" entre el Javascript y el HTML.

Todas estas partes son las que vamos a analizar en los próximos artículos con mayor detalle, comenzando por los decoradores, que introduciremos en el próximo artículo.

Este artículo es obra de *Alberto Basalo*
Fue publicado / actualizado en 01/08/2016
Disponible online en <https://desarrolloweb.com/articulos/concepto-teorico-componente-angular2.html>

Decorador de componentes en Angular 2

Qué es un decorador de componentes, qué función tiene y cómo se implementa en un componente básico de Angular 2.

Ahora, una de las funciones básicas que vas a tener que realizar en todo desarrollo con Angular

2 es la decoración de componentes. En sí, no es más que una declaración de cómo será un componente y las diversas piezas de las que consiste.

En el artículo de [introducción a los componentes](#) explicamos solo una de las partes que tiene el archivo .ts con el código Javascript / TypeScript principal de un componente. Lo que vimos hasta ahora era la clase que, decíamos, hacía las veces de controlador, que se exporta hacia afuera para que el flujo principal de ejecución de una aplicación sea capaz de conocer al componente. Además, contiene lo que se llama una función decoradora que conoceremos a continuación.



Qué es un decorador

Un decorador es una herramienta que tendremos a nuestra disposición en Javascript en un futuro próximo. Es una de las propuestas para formar parte del estándar ECMAScript 2016, conocido también como ES7. Sin embargo, ya están disponibles en TypeScript, por lo que podemos comenzar a usarlos ya en Angular 2.

Básicamente es una implementación de un patrón de diseño de software que en sí sirve para extender una función mediante otra función, pero sin tocar aquella original, que se está extendiendo. El decorador recibe una función como argumento (aquella que se quiere decorar) y devuelve esa función con alguna funcionalidad adicional.

Las funciones decoradoras comienzan por una "@" y a continuación tienen un nombre. Ese nombre es el de aquello que queramos decorar, que ya tiene que existir previamente. Podríamos decorar una función, una propiedad de una clase, una clase, etc.

Mira la primera línea del código del archivo .ts de tu componente principal.

```
import { Component } from '@angular/core';
```

Ese import nos está trayendo la clase Component. En la siguiente línea se decora a continuación, con el correspondiente "decorator". No es nuestro objetivo hablar sobre el patrón decorator en sí, ni ver las posibilidades de esta construcción que seguramente tendremos en el futuro ES7, así que vamos a centrarnos en lo que conseguimos hacer con Angular 2 mediante estos decoradores.

Nota: Uno de los motivos por los que Angular 2 ha tomado TypeScript como lenguaje es justamente por permitir usar decoradores. Con TypeScript podemos realizar la decoración

de código de ES7 ya mismo, lo que facilita la decoración del código.

Qué información se agrega por medio del decorador

Angular 2 usa los decoradores para registrar un componente, añadiendo información para que éste sea reconocido por otras partes de la aplicación. La forma de un decorador es la siguiente:

```
@Component({
  moduleId: module.id,
  selector: 'test-angular2-app',
  templateUrl: 'test-angular2.component.html',
  styleUrls: ['test-angular2.component.css']
})
```

Como apreciarás, en el decorador estamos agregando diversas propiedades específicas del componente. Esa información en este caso concreto se conoce como "anotación" y lo que le entregamos son unos "metadatos" (metadata) que no hace más que describir al componente que se está creando. En este caso son los siguientes:

- **moduleId**: esta propiedad puede parecer un poco extraña, porque siempre se le asigna el mismo valor en todos los componentes. Realmente ahora nos importa poco, porque no agrega ninguna personalización. Es algo que tiene que ver con CommonJS y sirve para poder resolver Urls relativas.
- **selector**: este es el nombre de la etiqueta nueva que crearemos cuando se procese el componente. Es la etiqueta que usarás cuando quieras colocar el componente en cualquier lugar del HTML.
- **templateUrl**: es el nombre del archivo .html con el contenido del componente, en otras palabras, el que tiene el código de la vista.
- **styleUrls**: es un array con todas las hojas de estilos CSS que deben procesarse como estilo local para este componente. Como ves, podríamos tener una única declaración de estilos, o varias si lo consideramos necesario.

Nota: Ese código de anotación o decoración del componente es generado por Angular CLI. Además, cuando creemos nuevos componentes usaremos el mismo Angular CLI para obtener el scaffolding (esqueleto) del cual partiremos. Por tanto, no hace falta que memorices la sintaxis para la decoración, porque te la darán hecha. En todo caso tendrás que modificarla si quieres cambiar el comportamiento del componente, los nombres de archivos del template (vista), hojas de estilo, etc.

De momento no necesitamos dar mucha más información sobre los decoradores. Es algo que debemos comenzar a usar para desarrollar componentes en Angular 2, pero no nos tiene que preocupar demasiado todavía, porque de momento no necesitaremos tocar mucho sobre ellos.

Era importante para finalizar nuestro análisis de la aplicación básica de Angular 2. Con esto creemos que debe quedar claro el código del primer componente que nos genera la aplicación de inicio creada por Angular CLI. Ahora que ya sabes lo básico, podemos continuar con otros

asuntos que seguro serán más entretenidos.

Este artículo es obra de *Miguel Angel Alvarez*
Fue publicado / actualizado en 09/08/2016
Disponible online en <https://desarrolloweb.com/articulos/decorador-componentes-angular2.html>

Crear un componente nuevo con Angular 2

En este artículo te vamos a enseñar a crear un nuevo componente con Angular CLI y luego a usarlo en tu aplicación Angular 2.

Después de nuestro repaso teórico a la figura de los [componentes en Angular 2](#) estamos en condiciones de irnos al terreno de lo práctico y ver cómo se generan nuevos componentes en una aplicación.

Si abres el index.html que hay en la carpeta "src" verás como en el body existe un único componente. Ahora, todos los que desarrollemos en adelante estarán dentro de ese primer componente. Esto no es nuevo, puesto que ya se comentó en el [Manual de Angular 2](#), pero está bien recordarlo para que quede claro. Partiremos de esta situación agregando ahora nuevos componentes con los que podremos expandir la aplicación.

Nota: En realidad nadie te obliga a tener un componente único como raíz. Podrías crear un componente y usarlo directamente en el index.html de tu aplicación. Haciendo esto convenientemente no habría ningún problema por ello. Aunque debido a que ese index.html es código generado y generalmente no lo queremos tocar, será más recomendable crear los componentes debajo del componente raíz.



Creamos un componente a través de Angular CLI

Ya usamos [Angular CLI](#) para generar el código de inicio de nuestra aplicación Angular 2. Ahora vamos a usar esta herramienta de línea de comandos para generar el esqueleto de un componente. Desde la raíz del proyecto lanzamos el siguiente comando:

```
ng generate component nombre-del-componente
```

Nota: Existe un alias para la orden "generate" que puedes usar para escribir un poco menos. Sería simplemente escribir "g".

```
ng g component nombre-del-componente
```

Además de componentes, la orden generate te permite crear el esqueleto de otra serie de artefactos como son directivas, servicios, clases, etc.

Reconociendo los archivos del componente

Si observas ahora la carpeta "src/app" encontrarás que se ha creado un directorio nuevo con el mismo nombre del componente que acabamos de crear. Dentro encuentras una serie de archivos que ya te deben de sonar porque los hemos analizado ya para el [componente inicial de las aplicaciones Angular2](#).

Ahora además queremos fijarnos en un archivo que se llama index.ts, que verás que solo tiene un export.

```
export { NombreDelComponenteComponent } from './nombre-del-componente.component';
```

Como ves, está exportando el propio componente, por su nombre. Ese componente es una clase (de OOP) que hay en "nombre-del-componente.component.ts", que a su vez se exportaba también.

```
export class NombreDelComponenteComponent implements OnInit {
```

En resumen, el archivo index.ts únicamente sirve para que, cuando importas un componente desde otro lugar de tu aplicación, no tengas que referirte al archivo "nombre-del-componente.component.ts" con todas sus letras, sino simplemente a la carpeta donde se encuentra.

Componente declarado para SystemJS

En todos los lugares donde, en adelante, deseemos usar ese componente tendremos que importarlo para que se conozca su código. Para ello hemos visto que solo tendremos que indicar el nombre del componente y no el archivo donde se escribió su clase.

Pero por debajo, para realizar todo lo que es la carga de módulos, se utiliza en una aplicación Angular 2 SystemJS. No nos hemos metido en profundidad con este sistema, pero básicamente lo que hace es traernos el código de aquello que necesitamos.

Dentro de system-config.ts se administran todas las librerías que podremos importar con

SystemJS. Entre ellas encontrarás que se hace una declaración del nuevo componente que acabamos de generar.

```
// App specific barrels.  
'app',  
'app/shared',  
'app/nombre-del-componente',
```

Esa declaración es la que realmente permite que, cuando importamos el componente, no tengamos que preocuparnos por la estructura de archivos que tiene, simplemente lo importamos con el nombre de la carpeta donde se encuentra.

Nota: Angular CLI es quien maneja y genera el código de system-config.ts, por lo que nosotros no tendremos que tocar nada generalmente. No obstante, queríamos mostrar este detalle para que no pasase desapercibido.

Javascript de nuestro componente

El archivo "nombre-del-componente.component.ts" contiene el código Javascript del componente.

Nota: Aprenderás que deberíamos decir que contiene el "código TypeScript del componente", dado que en realidad a día de hoy Angular CLI solo tiene la opción de generar código TypeScript. No debe representar un gran problema para ti, porque realmente todo código Javascript es también código [TypeScript](#), al que se le agregan ciertas cosas, sobre todo para el control de tipos.

Debes reconocer ya diversas partes:

- Imports de todo aquello que necesitemos. En principio de la librería @angular/core, pero luego veremos que aquí iremos colocando muchos otros imports a medida que vayamos necesitando código de más lugares.
- Decorador del componente, para su registro.
- Clase que hace las veces del controlador, que tengo que exportar.

Además ahora te pedimos simplemente echar un vistazo a la cabecera de la clase, en concreto a su "implements":

```
export class NombreDelComponenteComponent implements OnInit {
```

Ese implements es una interfaz, que no están disponibles todavía en Javascript, ni tan siquiera en ES6, pero que ya son posibles de usar gracias a TypeScript. Es simplemente como un contrato que dice que dentro de la clase del componente vamos a definir la función ngOnInit().

Sobre esa función no hablaremos mucho todavía, pero es un lugar donde podremos colocar código a ejecutar cuando se tenga la certeza que el componente ha sido inicializado ya.

Solo a modo de prueba, vamos a crear una propiedad llamada "dato" dentro de nuestro componente recién creado. El código nos quedará algo como esto:

```
import { Component, OnInit } from '@angular/core';
```

```
@Component({
  moduleId: module.id,
  selector: 'app-nombre-del-componente',
  templateUrl: 'nombre-del-componente.component.html',
  styleUrls: ['nombre-del-componente.component.css']
})
export class NombreDelComponenteComponent implements OnInit {

  dato = "Creando componentes para DesarrolloWeb.com";

  constructor() {}

  ngOnInit() {
    console.log('componente inicializado...');
  }
}
```

HTML del componente

El componente que acabamos de crear tiene un HTML de prueba, ya escrito en el archivo "nombre-del-componente.component.html".

Podemos agregarle la expresión para que se vea en el componente la propiedad que hemos generado del lado de Javascript, en la clase del componente. Tendrías algo como esto:

```
<p>
  nombre-del-componente works!
</p>
<p>
  {{ dato }}
</p>
```

Puedes abrir ese HTML y colocar cualquier cosa que consideres, simplemente a modo de prueba, para comprobar que consigues ver ese texto ahora cuando usemos el componente.

Nota: No hemos entrado todavía, pero seguro que alguno ya se lo pregunta o quiere saberlo. El componente tiene un archivo CSS (aunque se pueden definir varios en el decorador del componente) y podemos editarlo para colocar cualquier declaración de estilos. Veremos más adelante, o podrás comprobar por ti mismo, que esos estilos CSS se aplican únicamente al componente donde estás trabajando y no a otros componentes de la aplicación.

Con esto hemos terminado de explicar todo lo relativo a la creación de un componente. El componente está ahí y estamos seguros que estarás ansioso por verlo en funcionamiento en tu proyecto. Es algo que veremos ya mismo, en el próximo artículo del Manual de Angular 2.

Este artículo es obra de *Alberto Basalo*
Fue publicado / actualizado en 26/08/2016
Disponible online en <https://desarrolloweb.com/articulos/crear-componente-nuevo-angular2.html>

Usar un componente en Angular 2

Cómo se usan componentes creados por nosotros en Angular 2, una tarea que si bien es sencilla requiere de varios pasos.

En el pasado artículo realizamos todos los pasos para [crear un componente en Angular 2](#). Realmente vimos que la mayoría del código lo generas desde Angular CLI, lo que acelera mucho el desarrollo y facilita nuestra labor.

Ahora, para terminar nuestra práctica, vamos a aprender a usar el componente que acabamos de crear. Es una tarea sencilla, pero debido a la arquitectura de Angular y el modo de trabajo que nos marca, es algo que tendremos que realizar en varios pasos:



1. Crear el HTML para usar el componente

En el lugar de la aplicación donde lo vayas a usar el componente, tienes que escribir el HTML necesario para que se muestre. El HTML no es más que la etiqueta del componente, que se ha definido en la función decoradora, atributo "selector" ([como vimos al explicar los decoradores](#)).

```
<app-nombre-del-componente></app-nombre-del-componente>
```

Dado que estamos comenzando con Angular 2 y el anterior era el primer componente creado por nosotros mismos, solo lo podremos usar dentro del componente principal (aquel generado por Angular CLI al hacer construir el nuevo proyecto).

El HTML de este componente principal lo encontramos en el archivo "app.component.html".

En ese archivo, la vista del componente principal, debemos colocar la etiqueta para permitir mostrar el componente recién creado.

El problema es que esa etiqueta no es conocida por el navegador. Cuando creamos el componente generamos todo el código, en diversos archivos, necesario para dar vida al componente, pero habrá que incluirlo para que el navegador conozca esa etiqueta cuando se vaya a usar. Es lo que hacemos en los siguientes pasos.

2. Importar el código del componente

Como decíamos, desde el componente principal no conocemos el código del componente que se pretende usar. En Angular 2 en general, todo archivo con código Javascript que quieras usar, debe ser importado.

En el componente desde donde pretendas usar ese otro componente necesitas importar su código. Como verás en el archivo .ts principal de un componente, siempre comienza con un import de librerías que nos aporta Angular. Ahora, después del import de las librerías de Angular, colocaremos el import de nuestro componente:

```
import { NombreDelComponenteComponent } from './nombre-del-componente'
```

Ese import indica que te quieres traer la clase del componente y después del "from" está la ruta desde donde te la traes.

Nota: Recuerda que no necesitas decirle exactamente en qué fichero está la clase NombreDelComponenteComponent, porque existe un archivo index.ts en todo componente generado que hace de raíz y asocia el nombre de carpeta del componente al archivo .ts donde está la clase. Esto lo explicamos en el artículo anterior, cómo crear un componente, en el apartado "Reconociendo los archivos del componente".

3. Declarar que vas a usar este componente

El import permite conocer el código del componente, pero todavía no es suficiente para poder usarlo. Nos queda un último paso.

En el componente desde donde hemos incluido el import del componente, un poco más abajo encuentras la función decoradora del componente. En esa función debes declarar todos los componentes que pretendes usar.

Eso lo hago desde la función decoradora del componente principal y tengo que declarar todos los componentes y directivas a usar dentro del array "directives".

```
directives: [NombreDelComponenteComponent]
```

El decorador completo del componente principal se vería parecido a esto:


```
@Component({
  moduleId: module.id,
  selector: 'app-root',
  templateUrl: 'app.component.html',
  styleUrls: ['app.component.css'],
  directives: [NombreDelComponenteComponent]
})
```

Observa que el array "directives" me permite declarar que voy a usar varios componentes. Simplemente separo sus nombres por comas. Lo que indico, como podrás apreciar, es el nombre de la clase del componente que pretendo usar. Esa clase es la que has importado con el correspondiente "import" del paso anterior (punto 2).

Nota: Puede parecer un poco raro que el array se llame "directives" en vez de "components" y eso es porque en ese array podríamos declarar que vamos a usar tanto componentes como directivas. Además debes recordar, porque se mencionó en el artículo de teoría de los componentes en Angular 2, que un componente no es más que un tipo específico de directiva.

Y eso es todo! Al guardar los archivos se debería recargar de nuevo la aplicación en el navegador y deberías ver el HTML escrito para el componente que estás usando y que acabamos de crear.

El proceso puede parecer un tanto laborioso, pero a medida que se vaya repitiendo observaremos que no hay tanto trabajo. La parte más aburrida de escribir de un componente, el esqueleto o scaffolding, ya te lo dan hecho gracias a Angular CLI. Así que nos queda simplemente hacer la parte del componente que corresponde a las necesidades de la aplicación.

Con los conocimientos que hemos ido proporcionando en los anteriores capítulos del Manual de Angular 2, estamos seguros de que podrás colocar más código, tanto en el HTML como en la clase para proporcionar como práctica otros tipos de funcionalidad.

Este artículo es obra de *Alberto Basalo*
Fue publicado / actualizado en 31/08/2016
Disponible online en <https://desarrolloweb.com/articulos/usar-componente-angular2.html>

Lo básico de los módulos en Angular

Vamos ahora a abordar las bases de otro de los actores principales de las aplicaciones desarrolladas con Angular: los módulos. Un módulo (module en inglés) es una reunión de componentes y otros artefactos como directivas o pipes. Los módulos nos sirven principalmente para organizar el código de las aplicaciones Angular y por tanto debemos aprender a utilizarlos bien.

Trabajar con módulos en Angular

Cómo crear módulos, agrupaciones de componentes, directivas o pipes, en el framework Javascript Angular. Cómo crear componentes dentro de un módulo y cómo usarlos en otros modules.

Un módulo es uno de los elementos principales con los que podemos organizar el código de las aplicaciones en Angular. No deben ser desconocidos hasta este momento del [Manual de Angular](#), puesto que nuestra aplicación básica ya disponía de uno.

Sin embargo, en lugar de colocar el código de todos los componentes, directivas o pipes en el mismo módulo principal, lo adecuado es desarrollar diferentes módulos y agrupar distintos elementos en unos u otros. El orden se realizará de una manera lógica, atendiendo a nuestras propias preferencias, el modelo de negocio o las preferencias del equipo de desarrollo.

En este artículo aprenderás a trabajar con módulos, realizando operativas básicas como crear módulos y colocar componentes en ellos.

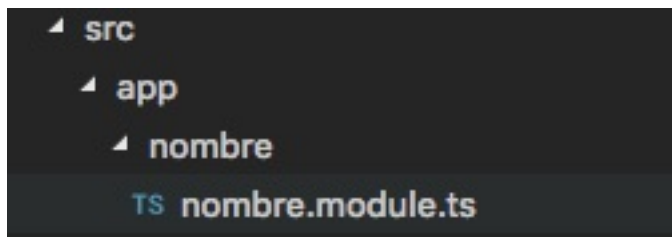


Crear un nuevo módulo

Para facilitar las tareas de creación de módulos nos apoyaremos en el [Angular CLI](#). El comando para generar ese módulo nuevo es "generate" y a continuación tenemos que indicar qué es lo que se quiere generar, en este caso "module", acabando con el nombre del módulo a crear.

```
ng generate module nombre
```

Una vez lanzado este comando en nuestro proyecto, dentro de la carpeta "src/app" se crea un subdirectorio con el mismo nombre del módulo generado. Dentro encontraremos además el archivo con el código del módulo.



Nota: tanto da que en el comando nombres el módulo como "nombre" o "Nombre" (con la primera en mayúscula). El CLI aplica las convenciones de nombres más adecuadas y como los módulos son clases, internamente les coloca en el código la primera letra siempre en mayúscula. Ya los nombres de los directorios y archivos es otra cosa y no se recomienda usar mayúsculas, por lo que los nombrará con minúscula siempre.

Ahora, si abrimos el código del módulo generado "nombre.module.ts", encontraremos cómo se define un módulo en Angular. La parte más importante es, como ya viene siendo habitual en Angular, un decorador. El decorador de los módulos se llama @NgModule.

```
@NgModule({  
  imports: [  
    CommonModule  
  ],  
  declarations: []  
})
```

Nota: este es el código generado de un módulo con el CLI para Angular 4. En tu caso puede tener algunas diferencias, dependiendo de la versión de Angular con la que estés trabajando.

Como ves en el decorador, tienes de momento un par de arrays definidos:

- imports: con los imports que este módulo necesita
- declarations: con los componentes, u otros artefactos que este module construye.

Generar un componente dentro del módulo

Ahora que tenemos nuestro primer módulo propio, vamos a agregar algo en él. Básicamente comenzaremos por añadirle un componente, usando como siempre el Angular CLI.

Hasta ahora todos los componentes que habíamos creado habían sido generados dentro del

módulo principal, pero si queremos podemos especificar otro módulo donde crearlos, mediante el comando:

```
ng generate component nombre/miComponente
```

Esto nos generará una carpeta dentro del módulo indicado, en la que colocará todos los archivos del componente recién creado.

```
└─ nombre
  └─ mi-componente
    # mi-componente.component.css
    <> mi-componente.component.html
    TS mi-componente.component.spec.ts
    TS mi-componente.component.ts
    TS nombre.module.ts
```

Nota: en este caso puedes observar como hemos colocado en el nombre del componente una mayúscula "miComponente", para separar palabras como en "camelCase". Por haberlo hecho así, Angular CLI ha nombrado el archivo separando las palabras por guiones "mi-componente.component.ts". Por su parte, podrás apreciar en el código que la clase del componente, se coloca con PascalCase, como mandan las guías de estilos para clases (class MiComponenteComponent).

Pero además, el comando del CLI también modificará el código del módulo, agregando automáticamente el import del componente y su referencia en el array "declarations". Ahora el código del módulo "nombre-modulo.module.ts" tendrá una forma como esta:

```
import { NgModule } from '@angular/core';
import { CommonModule } from '@angular/common';
import { MiComponenteComponent } from './micomponente/micomponente.component';

@NgModule({
  imports: [
    CommonModule
  ],
  declarations: [
    MiComponenteComponent
  ]
})
export class NombreModuloModule { }
```

Exportar del módulo hacia afuera

Más adelante, si queremos que este módulo exponga cosas hacia afuera, que se puedan llegar a utilizar desde otros módulos, tendremos que agregar una nueva información al decorador del

módulo: el array de exports.

Vamos a suponer que el componente "MiComponenteComponent" queremos que se pueda usar desde otros módulos. Entonces debemos señalar el nombre de la clase del componente en el array de "exports". Con ello el decorador del module quedaría de esta manera.

```
@NgModule({
  imports: [
    CommonModule
  ],
  declarations: [
    MiComponenteComponent
  ],
  exports: [
    MiComponenteComponent
  ]
})
```

Usar el componente en otros módulos

El último punto que nos queda por ver es cómo usar el componente MiComponenteComponent desde otros módulos. Para ello vamos a modificar manualmente el módulo principal de la aplicación, de modo que pueda conocer el componente definido en el módulo nuevo que hemos creado en este artículo.

Para importar el componente realmente lo que vamos a importar es el módulo entero donde se ha colocado, ya que el propio módulo hace la definición de aquello que se quiere exportar en "exports". Requiere varios pasos

Hacer el import del módulo con la sentencia import de Javascript

Para que Javascript (o en este caso TypeScript) conozca el código del módulo, debemos importarlo primero. Esto no es algo de Angular, sino del propio lenguaje en particular.

```
import { NombreModule } from './nombre/nombre.module';
```

Declarar el import en el decorador del module principal

La importación de nuestro módulo se realiza en la declaración "imports" del módulo principal.

Este es el código del decorador del módulo principal.

```
@NgModule({
  declarations: [
    AppComponent
  ],
  imports: [
    BrowserModule,
    NombreModule
  ],
  providers: [],
```

```
bootstrap: [AppComponent]
  })
```

Te tienes que fijar en el array imports, que tiene el módulo que hemos creado nosotros mismos en este artículo "NombreModule".

Usar el componente en HTML

Finalmente ya solo nos queda usar el componente. Para ello vamos a colocar en el template del componente raíz el selector declarado en el componente creado.

Nota: recuerda que el selector del componente es el tag o etiqueta que se debe usar para poder usar un componente. Esto ya se detalló anteriormente en este manual. Si no lo recuerdas o quieres más información lee el artículo [Decorador de componentes en Angular 2](#).

Así que simplemente abrimos el archivo "app.component.ts" y colocamos la etiqueta de nuestro nuevo componente generado.

```
<app-mi-componente></app-mi-componente>
```

Eso es todo, si servimos nuestra aplicación deberíamos ver el mensaje del componente funcionando, que de manera predeterminada sería algo como "mi-componente works!".

Si no lo vemos, o no vemos nada, entonces nos tenemos que fijar el error que nos aparece, que podría estar visible en la pantalla del terminal donde hemos hecho el "ng serve", o en la consola de Javascript de tu navegador. Posiblemente allí te diga que tal componente no se conoce, con un mensaje como "parse errors: 'app-mi-componente' is not a known element...". Eso quiere decir que no has hecho el import correctamente en el decorador del módulo principal. O bien que te has olvidado de hacer el exports del componente a usar, en el módulo recién creado.

Este artículo es obra de *Miguel Angel Alvarez*
Fue publicado / actualizado en 20/10/2017
Disponible online en <https://desarrolloweb.com/articulos/trabajar-modulos-angular.html>

Directivas esenciales de Angular

Vamos a conocer algunas directivas esenciales ofrecidas por el propio framework, para el desarrollo de las vistas en Angular.

Directiva ngClass en Angular 2

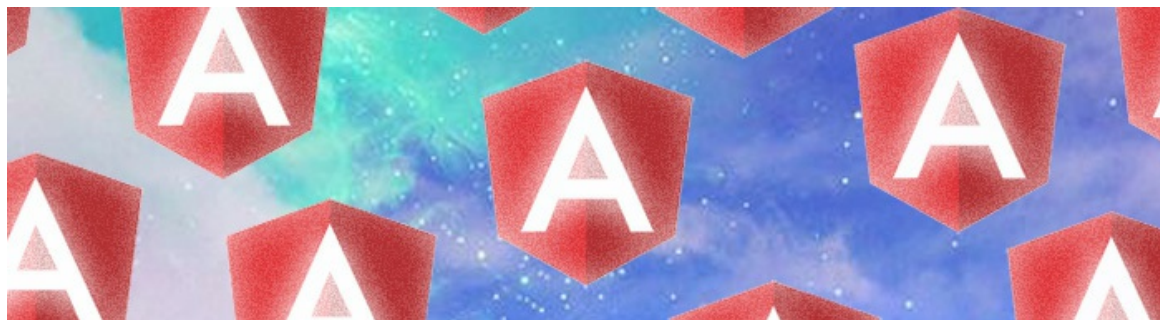
Estudio de la directiva ngClass de Angular 2, ejemplo práctico de un componente que usa esa directiva.

Después de varios artículos un tanto densos de en el [Manual de Angular 2](#) vamos a estudiar algunas cosas un poco más ligeras, que nos permitirán practicar con el framework sin añadir mucha complejidad a lo que ya conocemos. Para ello vamos a hacer una serie de pequeños artículos acerca de las directivas de Angular 2 más útiles en el día a día del desarrollo de aplicaciones.

Comenzamos con la directiva ngClass, que nos permite alterar las clases CSS que tienen los elementos de la página. Lo veremos en el marco del desarrollo de aplicaciones Angular con un ejemplo básico.

Si vienes de Angular 1.x verás que las directivas han perdido un poco de protagonismo en el desarrollo en Angular 2. Muchas de las antiguas directivas han desaparecido y su uso ha sido sustituido por otras herramientas diversas. Como norma ahora en Angular 2 las directivas se usarán para solucionar necesidades específicas de manipulación del DOM y otros temas estructurales.

Nota: Los componentes también pueden ser considerados como un tipo de directiva, aunque en este caso se usan para resolver problemas de negocio, como ya se introdujo en el artículo sobre las [características básicas de los componentes](#).



No todos los problemas de clases se necesitan resolver con ngClass

Lo primero es decir que la directiva `ngClass` no es necesaria en todos los casos. Las cosas más simples ni siquiera la necesitan. El atributo `"class"` de las etiquetas si lo pones entre corchetes funciona como propiedad a la que le puedes asignar algo que tengas en tu modelo. Esto lo vimos en el [artículo sobre la sintaxis de las vistas](#).

```
<h1 [class]="claseTitular">Titular</h1>
```

En este caso, `"claseTitular"` es una variable del modelo, algo que me pasa el controlador que tendrás asociado a un componente.

```
export class PruebaComponent implements OnInit {
  claseTitular: string = "class1";

  cambiaEstado() {
    this.claseTitular = "class2"
  }

  ngOnInit() {
  }
}
```

La class del `H1` valdrá lo que haya en la variable `claseTitular`. Cuando alguien llame al método `cambiaEstado()` se modificaría el valor de esa variable y por tanto cambiaría la clase en el encabezamiento.

Si esto ya resuelve la mayoría de las necesidades que se nos ocurren ¿para qué sirve entonces `ngClass`?

Asignar clases CSS con `ngClass`

La directiva `ngClass` es necesaria para, de una manera cómoda asignar cualquier clase CSS entre un grupo de posibilidades. Puedes usar varios valores para expresar los grupos de clases aplicables. Es parecido a como funcionaba en Angular 1.x.

A esta directiva le indicamos como valor:

1. Un array con la lista de clases a aplicar. Ese array lo podemos especificar de manera literal en el HTML.

```
<p [ngClass]="['negativo', 'off']">Pueden aplicarse varias clases</p>
```

O por su puesto podría ser el nombre de una variable que tenemos en el modelo con el array de clases creado mediante Javascript.

```
<p [ngClass]="arrayClases">Pueden aplicarse varias clases</p>
```

Ese array lo podrías haber definido del lado de Javascript.

```
clases = ['positivo', 'si'];
```

2. Un objeto con propiedades y valores (lo que sería un literal de objeto Javascript). Cada nombre de propiedad es una posible clase CSS que se podría asignar al elemento y cada valor es una expresión que se evaluará condicionalmente para aplicar o no esa clase.

```
<li [ngClass]="{positivo: cantidad > 0, negativo: cantidad < 0, off: desactivado, on: !desactivado}">Línea</li>
```

Ejemplo de aplicación de ngClass

Vamos a hacer un sencillo ejemplo de un componente llamado "BotonSino" que simplemente muestra un mensaje "SI" o "NO". Al pulsar el botón cambia el estado. Cada estado se representa además con una clase que aplica un aspecto.

Nota: No vamos a explicar las partes del componente porque se vieron con detalle en los artículos anteriores del Manual de Angular 2. Consultar toda la parte de desarrollo de componentes para más información.

Nuestro HTML es el siguiente:

```
<p>
  <button
    [ngClass]="{si: estadoPositivo, no: !estadoPositivo}"
    (click)="cambiaEstado()"
  >{{texto}}</button>
</p>
```

La etiqueta button tiene un par de atributos que son los que hacen la magia. Entre corchetes se aplica la directiva "ngClass", con un valor de objeto. Entre paréntesis se aplica el evento "click", con la invocación de la función encargada de procesar la acción. Además, el texto del botón es algo que nos vendrá de la variable "texto".

Nuestro Javascript será el siguiente:

```
import { Component, OnInit } from '@angular/core';

@Component({
  moduleId: module.id,
  selector: 'app-boton-sino',
  templateUrl: 'boton-sino.component.html',
  styleUrls: ['boton-sino.component.css']
})
export class BotonSinoComponent implements OnInit {

  texto: string = "SI";
  estadoPositivo: boolean = true;

  cambiaEstado() {
    this.texto = (this.estadoPositivo) ? "NO" : "SI";
  }
}
```

```
this.estadoPositivo = !this.estadoPositivo;
}

ngOnInit() {
}

}
```

Como sabes, este código TypeScript es la mayoría generado por Angular CLI. Lo que hemos hecho nosotros es lo que está dentro de la clase BotonSinoComponent.

En ella creamos las propiedades necesarias en la vista y el método que se encarga de procesar el cambio de estado.

Nuestro CSS:

Lógicamente, para que esto funcione necesitaremos declarar algunos estilos sencillos, al menos los de las clases que se usan en el HTML.

```
button {
  padding: 15px;
  font-size: 1.2em;
  border-radius: 5px;
  color: white;
  font-weight: bold;
  width: 70px;
  height: 60px;
}

.si{
  background-color: #6c5;
}

.no{
  background-color: #933;
}
```

Con esto es todo! Es un ejemplo muy sencillo que quizás para los que vienen de Angular 1.x resulte demasiado básico, pero seguro que lo agradecerán quienes estén comenzando con Angular en estos momentos. Más adelante lo complicaremos algo. Realmente la dificultad mayor puede ser [seguir los pasos para la creación del componente](#) y luego los [pasos para su utilización](#), pero eso ya lo hemos explicado anteriormente.

Este artículo es obra de *Miguel Angel Alvarez*
Fue publicado / actualizado en 13/09/2016
Disponible online en <https://desarrolloweb.com/articulos/directiva-nglass-angular2.html>

Directiva ngFor de Angular 2

Explicamos la directiva ngFor, o *ngFor, que nos permite repetir una serie de veces un bloque de HTML en aplicaciones Angular 2.

En este artículo vamos a conocer y practicar con una directiva de las más importantes en

Angular 2, que es la directiva ngFor, capaz de hacer una repetición de elementos dentro de la página. Esta repetición nos permite recorrer una estructura de array y para cada uno de sus elementos replicar una cantidad de elementos en el DOM.

Para los que vengan de Angular 1 les sonará, puesto que es lo mismo que ya se conoce de ngRepeat, aunque cambian algunas cosillas sobre la sintaxis para la expresión del bucle, así como algunos mecanismos como la ordenación.

Uso básico de ngFor

Para ver un ejemplo de esta directiva en funcionamiento estamos obligados a crear de antemano un array con datos, que deben ser enviados a la vista, para que ya en el HTML se pueda realizar esa repetición. Como todo en Angular 2 se organiza mediante un componentes, vamos a crear un componente que contiene lo necesario para poder usar esta el ngFor.



Comenzamos con el código de la parte de TypeScript, que es donde tendremos que crear los datos que estarán disponibles en la vista.

```
import { Component, OnInit } from '@angular/core';

@Component({
  moduleId: module.id,
  selector: 'app-listado-preguntas',
  templateUrl: 'listado-preguntas.component.html',
  styleUrls: ['listado-preguntas.component.css']
})
export class ListadoPreguntasComponent implements OnInit {

  preguntas: string[] = [
    "¿España ganará la Euro 2016?",
    "¿Hará sol el día de mi boda?",
    "¿Estás aprendiendo Angular 2 en DesarrolloWeb?",
    "¿Has hecho ya algún curso en EscuelaIT?"
  ];

  ngOnInit() {
  }

}
```

Este código nos debe de sonar, pues es el boilerplate de un componente (creado mediante Angular CLI) al que le hemos agregado la declaración de una propiedad de tipo array de strings.

Nota: Hemos asignado el valor de ese array de manera literal, pero lo normal sería que lo obtengas de alguna fuente como un servicio web, API, etc.

Luego, veamos el código HTML de este componente, que es donde colocamos la directiva ngFor.

```
<p *ngFor="let pregunta of preguntas">
  {{pregunta}}
</p>
```

Es algo muy sencillo, simplemente tenemos un párrafo que se repetirá un número de veces, una por cada elemento del array de preguntas. En este caso es un párrafo simple, pero si dentro de él tuviéramos más elementos, también se repetirían. Lo que tenemos que analizar con detalle es el uso de la directiva, aunque creemos que se auto-explica perfectamente.

```
*ngFor="let pregunta of preguntas"
```

Lo primero que verás es un símbolo asterisco (*) que quizás parezca un poco extraño. No es más que "azúcar sintáctico" para recordarnos que estas directivas (las comenzadas por el asterisco) afectan al DOM, produciendo la inserción, manipulación o borrado de elementos del mismo.

Nota: En las explicaciones que encontrarás en la documentación de Angular 2 sobre el origen del asterisco en el nombre de la directiva nos mencionan detalles acerca de su implementación a bajo nivel, indicando que para ello se basan en el tag [TEMPLATE, uno de las especificaciones nativas de Web Components](#).

Como valor de la directiva verás que se declara de manera interna para este bucle una variable "pregunta", que tomará como valor cada uno de los valores del array en cada una de sus repeticiones.

Nota: "let" es una forma de declarar variables en Javascript ES6. Quiere decir que aquella variable sólo tendrá validez dentro del bloque donde se declara. En este caso "de manera interna" nos referimos a que "pregunta" solo tendrá validez en la etiqueta que tiene el ngFor y cualquiera de sus los elementos hijo.

Recorrido a arrays con objetos con ngFor

Generalmente ngFor lo usarás para recorrer arrays que seguramente tendrán como valor en cada una de sus casillas un objeto. Esto no cambia mucho con respecto a lo que ya hemos visto en este artículo, pero sí es una bonita oportunidad de aprender algo nuevo con TypeScript.

De momento veamos las cosas sin TypeScript para ir progresivamente. Así sería cómo quedaría la declaración de nuestro array, al que todavía no indicaremos el tipo para no liarnos.

```
preguntasObj = [  
  {  
    pregunta: "¿España ganará la Euro 2016?",  
    si: 22,  
    no: 95  
  },  
  {  
    pregunta: "¿Estás aprendiendo Angular 2 en DesarrolloWeb??",  
    si: 262,  
    no: 3  
  },  
  {  
    pregunta: "¿Has hecho ya algún curso en EscuelaIT??",  
    si: 1026,  
    no: 1  
  }  
]
```

Como ves, lo que antes era un array de strings simples ha pasado a ser un array de objetos. Cada uno de los objetos nos describen tanto la pregunta en sí como las respuestas positivas y negativas que se han recibido hasta el momento.

Ahora, al usarlo en la vista, el HTML, podemos mostrar todos los datos de cada objeto, con un código que podría ser parecido a este:

```
<p *ngFor="let objPregunta of preguntasObj">  
  {{objPregunta.pregunta}}:  
  <br>  
  <span class="si">Si {{objPregunta.si}}</span> /  
  <span class="no">No {{objPregunta.no}}</span>  
</p>
```

Como ves en este código, dentro del párrafo tengo acceso a la pregunta, que al ser un objeto, contiene diversas propiedades que uso para mostrar los datos completos de cada item.

Modificación implementando Interfaces TypeScript

Como has visto, no existe mucha diferencia con respecto a lo que teníamos, pero ahora vamos a darle un uso a TypeScript que nos permitirá experimentar algo que nos aporta el lenguaje: interfaces.

En este caso vamos a usar las interfaces simplemente para definir un tipo de datos para las preguntas, un esquema para nuestros objetos pregunta. En este caso solo lo vamos a usar para que, a la hora de escribir código, el editor nos pueda ayudar indicando errores en caso que la interfaz no se cumpla. Así, a la hora de escribir código podremos estar seguros que todas las preguntas con las que trabajemos tengan los datos que son necesarios para la aplicación.

Nota: Las interfaces que se sabe son mecanismos para solventar las carencias de herencia múltiple, en este caso las vamos a usar como una simple definición de tipos.

Algo tan sencillo como esto:

```
interface PreguntasInterface {  
  pregunta: string;  
  si: number;  
  no: number;  
}
```

Permite a TypeScript conocer el esquema de un objeto pregunta. Ahora, apoyándonos en esa interfaz podrás declarar tu array de preguntas de esta manera.

```
preguntasObj: PreguntasInterface[] = [  
  {  
    pregunta: "¿Te gusta usar interfaces?",  
    si: 72,  
    no: 6  
  }  
]
```

Ahora estamos indicando el tipo de los elementos del array, diciéndole que debe concordar con lo definido en la interfaz. ¿Te gusta? Quizás ahora no aprecies mucha diferencia, pero esto se puede usar para varias cosas, significando una ayuda en la etapa de desarrollo, y sin afectar al rendimiento de la aplicación, puesto que las interfaces en TypeScript una vez transpilado el código no generan código alguno en Javascript.

Nota: Lo normal es que coloques el código de la interfaz en un archivo independiente y que hagas el correspondiente "import". Recordando que Angular CLI tiene un comando para generar interfaces que te puede resultar útil. De momento si lo deseas, a modo de prueba lo puedes colocar en el mismo archivo que el código TypeScript del componente.

Quizás para los que no estén acostumbrados a TypeScript sea difícil hacerse una idea exacta sobre cómo te ayudaría el editor de código por el simple hecho de usar esa interfaz. Para ilustrarlo hemos creado este vídeo en el que mostramos las ayudas contextuales cuando estamos desarrollando con Visual Studio Code.

Para ver este vídeo es necesario visitar el artículo original en:
<https://desarrolloweb.com/articulos/directiva-ngfor-angular2.html>

Hablaremos más sobre interfaces en otras ocasiones. Ahora el objetivo era simplemente ver una pequeña muestra de las utilidades que podría aportarnos.

Como habrás visto no es difícil entender esta directiva, pero ten en cuenta que hemos visto lo esencial sobre las repeticiones con ngFor. Hay mucho más que habría que comentar en un

futuro, acerca de usos un poco más avanzados como podría ser la ordenación de los elementos del listado.

Este artículo es obra de *Miguel Angel Alvarez*
Fue publicado / actualizado en 27/09/2016
Disponible online en <https://desarrolloweb.com/articulos/directiva-ngfor-angular2.html>

Directiva ngModel

Explicaciones sobre la directiva ngModel de Angular, con ejemplos de uso diversos, con binding de una y dos direcciones. Importar FormsModule para su funcionamiento en Angular 4.

La directiva ngModel es un viejo conocido para las personas que vienen de las versiones antiguas del framework, cuando se llamaba AngularJS. Quizás para ellos no requiera tantas explicaciones de concepto, aunque sí será importante explicar cómo usarla, porque han cambiado bastantes cosas.

De todos modos, tanto para desarrolladores experimentados como para los más novatos, vamos a repasar en este artículo del [Manual de Angular](#) los aspectos básicos de ngModel y ver algunos ejemplos sencillos de funcionamiento en componentes.



Qué es ngModel

Pensando en las personas que son nuevas en Angular, tendremos que comenzar aclarando qué es ngModel. Básicamente se trata de un enlace, entre algo que tienes en la definición del componente con un campo de formulario del template (vista) del componente.

Nota: "model" en ngModel viene de lo que se conoce como el modelo en el MVC. El modelo trabaja con los datos, así que podemos entender que el enlace creado con ngModel permite que desde la vista pueda usar un dato. Sin embargo, con expresiones ya se podía usar un dato, volcando su valor en la vista como {{dato}}. La diferencia es que al usar ese dato en campos de formulario, debes aplicar el valor mediante la directiva ngModel.

Por ejemplo, tenemos un componente llamado "cuadrado", que declara una propiedad llamada "lado". La class del componente podría quedar así:

```
export class CuadradoComponent {  
  lado = 4;  
}
```

Si queremos que ese valor "lado" se vuelque dentro de un campo INPUT de formulario, tendríamos que usar algo como esto.

```
<input type="number" [ngModel]="lado">
```

Estamos usando la directiva ngModel como si fuera una propiedad del campo INPUT, asignándole el valor que tenemos declarado en el componente.

Dar soporte a la propiedad ngModel en el campo de formulario

Sin embargo, nuestro componente "cuadrado" no funcionaría todavía, porque Angular 4 en principio no reconoce ngModel como propiedad de un campo INPUT de formulario. Por ello, si ejecutas tu aplicación con el código tal como hemos hecho hasta ahora, obtendrás un mensaje de error como este: "Can't bind to 'ngModel' since it isn't a known property of 'input'."

La solución pasa por traernos esa propiedad, que está en el módulo "FormsModule".

Para ello tenemos que hacer la operativa tradicional de importar aquello que necesitamos. En este caso tendremos que importar FormsModule en el módulo donde vamos a crear aquel componente donde se quiera usar la directiva ngModule.

Por ejemplo, si nuestro componente "CuadradoComponent" está en el módulo "FigurasModule", este sería el código necesario para declarar el import de "FormsModule".

```
import { NgModule } from '@angular/core';  
import { CommonModule } from '@angular/common';  
import { CuadradoComponent } from './cuadrado/cuadrado.component';  
import { FormsModule } from '@angular/forms';  
  
@NgModule({  
  imports: [  
    CommonModule,  
    FormsModule  
  ] ,  
  declarations: [CuadradoComponent],  
  exports: [CuadradoComponent]  
})  
export class FigurasModule { }
```

Del código del módulo anterior, debes fijarte en dos cosas:

1.- El import de FormsModule, que viene de @angular/forms

```
import { FormsModule } from '@angular/forms';
```

2.- La declaración en el array de imports del módulo en cuestión:

```
imports: [  
  [...]  
  FormsModule  
],
```

Con esta infraestructura ya somos capaces de usar ngModule en las vistas de los componentes Angular.

Enlace de una o de dos direcciones con ngModel

Tal como hemos dejado el código hasta el momento en la vista, el input estaba asociado al valor de una propiedad del componente, sin embargo, era un enlace de una única dirección.

```
<input type="number" [ngModel]="lado">
```

Con esa sintaxis en la vista, lo que haya en la propiedad "lado" se escribía como valor del input, pero no encontramos el doble binding. Y aunque esto es algo que ya se trató en el artículo de [sintaxis de las vistas en Angular](#), queremos recordar que, para especificar el doble binding, se debe usar la sintaxis "banana in a box".

```
<input type="number" [(ngModel)]="lado">
```

Ahora, lo que se escriba en el campo INPUT también viajará hacia el modelo, actualizando el valor de la propiedad "lado" del componente.

Evento ngModelChange

Si lo deseas, también tienes disponible un evento llamado "ngModelChange" que se ejecuta cuando cambia el valor en la propiedad asociada a un ngModel, con el que podríamos conseguir un comportamiento idéntico al visto en el punto anterior del doble binding, pero sin usar el binding doble.

Tienes que trabajar con ngModelChange en conjunto con la directiva ngModel. Mediante ngModel asocias la propiedad que quieres asociar y entonces tendrás la posibilidad de asociar un manejador de evento a ngModelChange, cada vez que esa propiedad cambia.

Dentro del manejador de evento podemos además usar una variable llamada \$event, en la que recibimos el nuevo valor escrito, que podríamos volcarla de nuevo a la propiedad por medio de una asignación, para conseguir el mismo efecto del binding en las dos direcciones. El código te quedaría como esto:

```
<input type="number" [ngModel]="lado" (ngModelChange)="lado = $event">
```

Nota: Esta sintaxis no aporta ninguna ventaja en términos de rendimiento, por lo que en principio no se suele usar mucho. Generalmente vamos a preferir usar la sintaxis del binding de dos direcciones [(ngModel)], por ser más concisa. De todos modos, podría ser interesante disponer de ngModelChange en el caso que, cuando cambiase el modelo, necesitases realizar otras acciones, adicionales a la simple asignación de un nuevo valor en la propiedad del componente.

Ejemplo realizado en este artículo

Ahora para la referencia, voy a dejar el código realizado para ilustrar el comportamiento y uso de la directiva ngModel. Estará compuesto por varios archivos.

Vista del componente:

Comenzamos con la vista del componente, archivo cuadrado.component.html, que es lo más importante en este caso, ya que es el lugar donde hemos usado ngModel.

He creado varias versiones de enlace al input, como puedes ver a continuación.

```
<?>
  Tamaño del lado {{lado}}
</?>
<?>
  1 way binding <input type="number" [ngModel]="lado">
</?>
<?>
  2 way binding: <input type="number" [(ngModel)]="lado">
</?>
<?>
  Evento ngModelChange: <input type="number" [ngModel]="lado" (ngModelChange)="cambiaLado($event)">
</?>
```

Otra cosa que apreciarás en la vista es que el evento ngModelChange no tiene escrito el código del manejador en la propia vista, por considerarlo un antipatrón. Es más interesante que el código se coloque dentro de un método del componente. En este caso tendrás que enviarle al método el valor \$event, para que lo puedas usar allí.

Código TypeScript del componente:

El código del componente estará en el archivo cuadrado.component.ts y no tiene ninguna complicación en especial.

```
import { Component } from '@angular/core';

@Component({
  selector: 'dw-cuadrado',
```



```

templateUrl: './cuadrado.component.html',
styleUrls: ['./cuadrado.component.css']
})
export class CuadradoComponent {
  lado = 1;

  cambiaLado(valor) {
    this.lado = valor;
  }
}

```

Módulo donde se crea el componente:

Por último recuerda que es esencial que importes en el módulo donde estés creando este componente el propio módulo del core de Angular donde se encuentra la directiva NgModule. Esto lo hemos descrito en un bloque anterior, pero volvemos a colocar aquí el código fuente:

```

import { NgModule } from '@angular/core';
import { CommonModule } from '@angular/common';
import { CuadradoComponent } from './cuadrado/cuadrado.component';
import { FormsModule } from '@angular/forms';

@NgModule({
  imports: [
    CommonModule,
    FormsModule
  ],
  declarations: [CuadradoComponent],
  exports: [CuadradoComponent]
})
export class FigurasModule { }

```

Por supuesto, para poder usar este componente "CuadradoComponent" en otro módulo tendrás que hacer el correspondiente import, pero esto es algo que ya se trató en el artículo de los [módulos en Angular](#).

Nota: Además de la aplicación de ngModel que hemos conocido en este artículo, relacionada directamente con el sistema de binding de Angular, ngModel también sirve para definir qué campos de formulario deben ser tratados al generarse los objetos ngForm. Estos objetos se crean automáticamente y Angular los pone a disposición para poder controlar el formulario de una manera muy precisa. Ese uso lo veremos más adelante cuando hablemos en detalle de los formularios de Angular.

Este artículo es obra de *Miguel Angel Alvarez*
 Fue publicado / actualizado en 26/10/2017
 Disponible online en <https://desarrolloweb.com/articulos/directiva-ngmodel-angular.html>

Binding en Angular al detalle

En los siguientes artículos vamos a explorar detenidamente el sistema de data-binding creado con Angular, tratando con profundidad algunos temas ya usados dentro de este manual, así como abordando nuevos temas importantes relacionados con el bindeo de datos.

Interpolación `{{}}` en Angular al detalle

Todo lo que tienes que saber sobre el binding por interpolación de strings en Angular, generada con la sintaxis de las dobles llaves `{{}}`.

Con este artículo comenzamos una serie de entregas del [Manual de Angular](#) en las que vamos a abordar distintos aspectos del binding en este framework. Son importantes, porque a pesar de ser cosas básicas muchas veces las dejamos pasar y ese desconocimiento acaba creando confusiones más adelante.

De todos modos, no vamos a explicar todo desde cero, ya que las bases del data-binding ya las hemos conocido en artículos anteriores del manual. Concretamente es interesante que hayas leído el artículo sobre la [sintaxis para las vistas de componentes](#).

En este artículo vamos a tratar muchas cosas sobre la interpolación en Angular que quizás hemos dado por sentadas en artículos anteriores y que merece la pena tratar de manera detallada y clara.



Qué es la interpolación de strings

La interpolación de cadenas, también conocida en inglés como "string interpolation", o simplemente como interpolación, es un mecanismo de Angular de sustitución de una expresión por un valor de cadena en un template.

Cuando Angular ve en un template algo escrito entre dobles llaves `{{}}` lo evalúa y lo trata de convertir en una cadena, para luego volcarlo en el template.

```
<p>Esto es un caso de interpolación de {{algunaCadena}}</p>
```

Eso quiere decir a Angular que debe sustituir una propiedad del componente llamada "algunaCadena" y colocar su valor tal cual en el template.

Si "algunaCadena" es una cadena, simplemente se escribirá su valor en el template, pero si eso no era una cadena tratará de colocarlo de manera que lo fuera. Por ejemplo, si tuviera un valor numérico colocará el número tal cual en el template.

Puedes usar también la interpolación como valor de propiedades de elementos HTML, como es el siguiente caso.

```

```

En este caso se colocará el valor de la propiedad urlImagen como src para el elemento IMG.

La interpolación es dinámica. Quiere decir que, si cambia el valor de la propiedad del componente, Angular se dará el trabajo de cambiar todos los lugares donde se está haciendo uso de esa propiedad, lo que cambiaría el texto que hay escrito del párrafo anterior, o cambiaría la imagen que se está visualizando en el elemento IMG anterior.

Expresiones, entre las dobles llaves

Lo que se coloca entre las dobles llaves son llamadas expresiones. Podemos crear expresiones simples y complejas y Angular se dará el trabajo de evaluarlas antes de volcar el resultado dentro del template.

Lo que es importante es que, aquella evaluación de la expresión, debe de ser convertida en una cadena antes de volcarse en un template. Es decir, cualquier expresión al final de cuentas se convertirá en una cadena y eso es lo que se colocará en la vista del componente.

Una expresión puede ser algo tan simple como una propiedad del componente, como las usadas anteriormente en este artículo.

```
{{algunaCadena}}
```

Esa sencilla expresión se evalúa al valor que tenga la propiedad algunaCadena y se vuelca en el template.

También podemos ver expresiones con operaciones matemáticas.

```
{{ 1+ 1 }}
```

Expresiones con operador lógico de negación:

```
{{ ! valorBoleano }}
```

Incluso expresiones cuyo valor es calculado por un método del componente.

```
{{ metodoComponente() }}
```

Lo que devuelva ese método del componente es lo que se colocará en el template. Ese método se volverá a ejecutar cada vez que el estado del componente cambie, es decir, cada vez que cambie una de sus propiedades, produciendo siempre una salida actualizada.

Nota: aunque vamos a insistir sobre este punto y se entenderá mejor si sigues la lectura del artículo, hay que decir que ese método tiene que limitarse a producir salida y además ser sencillo y rápido de ejecutar. Esto evitará afectar negativamente al rendimiento de las aplicaciones. el motivo es sencillo: si durante la ejecución de la aplicación se modifican las propiedades del componente, Angular volverá a ejecutar ese método, actualizando la salida convenientemente. Eso se hará ante cualquier pequeño cambio en el componente, y no solo ante cambios en las propiedades con las que trabaje el método, incluso se producirá solamente por haber ocurrido un evento susceptible de modificar el estado. Si pones en el método código que tarde en ejecutarse, Al producirse muchas invocaciones repetidas a este método, se multiplicará el coste de tiempo de ejecución, produciendo que la aplicación caiga en rendimiento y afectando negativamente a la experiencia de usuario.

Los efectos laterales están prohibidos en las expresiones

La interpolación en Angular es un enlace (binding) de una única dirección. Cuando cambian los valores en el componente viajan hacia el template, produciendo la actualización de la vista.

La interpolación no debería producir cambios en la otra dirección, es decir, modificar algo en la vista afectando al estado del componente. Dicho de otro modo, las expresiones nunca deben contener sentencias que puedan producir efectos laterales. Es decir, código que pueda afectar a cambios en las propiedades de componentes.

Ten en cuenta que las expresiones tienen como utilidad devolver una cadena para volcarla en el template del componente, por lo que no deberías colocar lógica de negocio en ellas, manipulando variables o propiedades de estado del componente o la aplicación.

Por este detalle, algunos operadores están prohibidos en las expresiones, por ejemplo los de asignación, incrementos, decrementos, etc. Por ejemplo, nunca hagas algo como esto:

```
{{ valorBoleano = false }}
```

Si Angular observa una expresión con operaciones capaces de realizar efectos laterales, te devolverá un error, como el que ves en la siguiente imagen:

```
✖ ▶ Uncaught Error: Template parse errors: compiler.es5.js:1694
Parser Error: Bindings cannot contain assignments at column
17 in [

{{ valorBoleano = false }}]
```

Por este mismo motivo, cuando dentro de una expresión invocas a un método, no deberías modificar en el código de ese método datos del componente capaces de producir efectos laterales, sino simplemente limitarte a producir una salida.

```
{{ metodoProduceEfectosLaterales() }}
```

En el caso que `metodoProduceEfectosLaterales()` tenga código que produzca cambios el el estado del componente Angular no te lo va a advertir y va a ejecutar el método sin producir ningún mensaje de error, pero aun así no lo deberías hacer. No es solo por ceñirse a las buenas prácticas, sino porque Angular no tendrá en cuenta aquel cambio en el estado del componente para desencadenar otras operaciones necesarias, lo que puede producir que tus templates no muestran los cambios en propiedades u otros efectos no deseables.

Otras consideraciones en las expresiones

Otras cosas que está bien saber cuando se usan expresiones son las siguientes:

- Las expresiones tienen como contexto al componente del template que se está desarrollando. Pero no pueden acceder al contexto global, como variables globales u objetos como `document` o `window`. Tampoco a cosas del navegador como `console.log()`.
- No se pueden escribir estructuras de control, como sentencias `if`, `for`, etc.
- La ejecución de una expresión debe ser directa. Por ejemplo no podrás hacer que se consulte un API REST en una expresión.
- Mantén la simplicidad. Cosas como usar una negación en la expresión es correcto, pero si lo que tienes que colocar es cierta lógica compleja, entonces conviene usar un método en el componente.
- Ofrecer siempre el mismo resultado con los mismos valores de entrada. Esto se conoce como expresiones idempotentes, aquellas que ejecutadas varias veces dan siempre el mismo resultado. Obviamente, si la entrada es distinta y los datos que se usan para calcular el valor de la expresión cambian, el resultado también cambiará, pero seguirá siendo idempotente si con el mismo juego de valores de entrada conseguimos siempre la misma salida. Esto es necesario para que los algoritmos de detección de cambios de Angular funcionen correctamente.

Este artículo es obra de *Miguel Angel Alvarez*
Fue publicado / actualizado en 30/10/2017
Disponible online en <https://desarrolloweb.com/articulos/binding-interpolacion-angular.html>

Binding a propiedades en Angular al detalle

Qué es el property binding, junto con una serie de detalles importantes sobre el bindeo a propiedades que como desarrollador Angular debes conocer.

Aunque ya hemos hablado del binding de propiedades, vamos a volver sobre este concepto pues hemos pasado por alto bastantes detalles en los que merece la pena detenerse con calma. Se trata de una característica disponible en el [desarrollo de templates de los componentes](#), que hemos visto aplicada a lo largo varios ejemplos del [Manual de Angular](#), pero esperamos poder ofrecer algunos conocimientos extra que seguramente te vendrán bien.

Comenzaremos repasando de nuevo el concepto, para ofrecer luego diversas explicaciones adicionales sobre cómo funciona el property binding. Además resolveremos algunas dudas comunes de las personas que se inician en Angular.



Qué es el bindeo a propiedades

El binding a propiedades, o "property binding" en inglés, sirve para asignar un valor a una propiedad de un elemento de un template. Esa asignación podrá ser un valor literal, escrito tal cual en el template, pero generalmente se tratará de un valor obtenido a través de una propiedad del componente, de modo que si el estado del componente cambia, también cambie la propiedad del elemento asignada en el template.

Para que no quede lugar a dudas tenemos que definir exactamente qué es una propiedad. De hecho, en el párrafo anterior hemos usado la palabra "propiedad" para referirnos a dos cosas, lo que puede dar mayor pie a confusión.

Propiedades de componentes

Los componentes generados con Angular se implementan mediante una clase, de programación orientada a objetos. Las propiedades de esa clase, los datos que almacena el componente, son lo que llamamos "propiedades del componente".

Generalmente en este artículo cuando nos referimos a "property binding" no nos referimos específicamente a propiedades del componente, sino a propiedades expresadas en un template. Éstas pueden ser propiedades de elementos HTML (propiedades del DOM de esos elementos) o propiedades de componentes en general.

En el HTML del componente, lo que llamamos el template o la vista, las propiedades son cualquier cosa a las que asignamos valores por medio de binding. Esto lo podemos ver bien con un ejemplo.

A continuación tienes elementos del HTML a los que definimos valores por medio de atributos.

```
  
<button disabled>Estoy desactivado</button>
```

A esos atributos de las etiquetas les asignamos valores, que sirven para inicializar los elementos HTML. Debes de saber de sobra cómo funcionan.

Tal como están, esa etiqueta IMG o ese botón tendrán siempre el mismo valor en sus propiedades "src" y "disabled". Sin embargo, si quisieras asignar un valor dinámico a uno de esos atributos, tomando algo definido mediante una propiedad del componente, tendrías que acudir al property binding. Se define el bindeo a propiedad mediante una sintaxis especial de Angular, asunto de este artículo.

```
<img [src]="rutaImagen">  
<button [disabled]="estadoBoton">Estoy activado o desactivado</button>
```

En este caso es donde encontramos las propiedades genéricas a las que nos hacemos referencia, aquellas expresadas en el template con la notación de los corchetes de inicio y de cierre.

Gracias al binding de propiedades tenemos una imagen y un botón igualmente, pero en la imagen su src se calcula en función de la propiedad del componente llamada "rutaImagen". En el caso del botón, el estado activo o desactivado, está definido por la propiedad del componente "estadoBoton".

Lo importante que debemos entender es: cuando trabajas con propiedades del template, no estás asignando valores a atributos de HTML. De hecho, al ponerle los corchetes dejan de ser atributos del HTML para pasar a ser propiedades del template de Angular.

Variantes del property binding

Mediante los corchetes puedes asignar valores dinámicos a propiedades de un componente, propiedades de directivas, o en el caso de los elementos HTML nativos, estás asignando valores directamente al DOM.

Vamos a analizar más de cerca estos casos con varios ejemplos.

```
<img [src]="rutaImagen">
```

En este ejemplo estamos colocando un valor a una propiedad de un elemento nativo del HTML, que nos acepta un string.

Sin embargo no todas las propiedades aceptan cadenas, algunas aceptan booleanos, como era el caso del botón.

```
<button [disabled]="estadoBoton">Estoy activado o desactivado</button>
```

Este es un caso interesante, porque el elemento nativo del HTML funciona de modo diferente. Si tiene el atributo "disabled" está desactivado. Si ese atributo no aparece, entonces estará activado. Ya para definir el valor de la propiedad de Angular aparecerá siempre el atributo, pero en algunas ocasiones estará desactivado y en otras activado, dependiendo del valor de la variable booleana "estadoBoton".

También podemos encontrar binding a propiedades en una directiva, como se puede ver en el siguiente código.

```
<div [ngStyle]="objEstilos">DIV con estilos definidos por Angular</div>
```

En este caso, mediante la propiedad "objEstilos" se está definiendo dinámicamente el style de esta división. Si cambia el objeto cambiará la aplicación de los estilos al elemento.

Por último, tenemos asignación a propiedades personalizadas de nuestros propios componentes.

```
<mi-componente [propiedadPersonalizada]="valorAsignado"></mi-componente>
```

Este es un caso especial, que explicaremos con detalle cuando hablemos de las propiedades @Input de componentes. Básicamente nos permiten crear cualquier tipo de propiedad en componentes, que somos capaces de definir en el padre. En ese caso concreto, mediante la propiedad del componente padre "valorAsignado" estamos aplicando el valor del componente hijo en la propiedad "propiedadPersonalizada".

Binding de una dirección

Como se ha mencionado, la asignación de una propiedad es dinámica. Es decir, si cambia con el tiempo, también cambiará la asignación al elemento al que estamos bindeando.

Volvemos de nuevo al ejemplo:

```
<img [src]="rutaImagen">
```

Al crearse la imagen se tomará como valor de su "src" lo que haya en la propiedad del componente "rutaImagen". Si con el tiempo "rutaImagen" cambia de valor, el valor viajará también a la imagen, alterando el archivo gráfico que dicha imagen muestra.

Property binding es siempre de una dirección, de padre a hijo. El padre define un valor y lo asigna al hijo. Por tanto, aunque el hijo modificase el valor nunca viajará al padre.

Obviamente, asignando un valor a un [src] de una imagen dicha imagen no va a cambiar el valor, pero en el caso de un componente creado por nosotros sí que podría ocurrir.

```
<app-cliente [nombre]="nombreCliente"></app-cliente>
```

En este caso, el padre ha definido que el nombre del cliente sea lo que éste tiene en su propiedad "nombreCliente". Podría ocurrir que el componente app-cliente cambiase el valor de su propia propiedad "nombre". Si eso ocurre, el componente padre no notará nada por darse un binding de una única dirección (padres a hijos).

Nota: En breve explicaremos cómo es posible crear componentes que acepten valores de entrada en sus propiedades. Esto adelantamos que se hace con el decorador @Input al definir la propiedad en la clase del componente.

Valores posibles para una propiedad

Entre las comillas asignadas como valor a una propiedad podemos colocar varias cosas. Lo común es que bindeemos mediante una propiedad del componente, tal como se ha visto en ejemplos anteriores, pero podríamos colocar otro código TypeScript.

Así, dentro del valor de una propiedad, podemos usar un pequeño conjunto de operadores como el de negación "!", igual que teníamos en las expresiones en el binding por interpolación de cadenas.

Además vamos a ver un ejemplo interesante porque nos dará pie a explicar otras cosas: podríamos bindear a un literal.

```
<img [src]="ruta.jpg">
```

En este caso "ruta.jpg" está escrito entre comillas simples, luego es un literal de string. Quiere decir que Angular lo evaluará como una cadena, lo asignará a la propiedad y se olvidará de él. Esto no tiene mucho sentido, ya que lo podríamos haber conseguido exactamente el mismo efecto con el propio atributo HTML, sin colocar los corchetes.

```

```

Esta posibilidad (la asignación de un literal) tendrá más sentido al usarlo en componentes personalizados.

```
<app-cliente nombre="DesarrolloWeb.com"></app-cliente>
```

Al no colocar la propiedad entre corchetes, es como hacer un binding a un literal de string. Por tanto, el componente app-cliente recibirá el valor "DesarrolloWeb.com" en su propiedad "nombre" al inicializarse, pero no se establecerá ningún binding durante su existencia.

También podemos bindear a un método, provocando que ese método se ejecute para aplicar un valor a la propiedad bindeada.

```
<p [ngClass]="obtenClases()">Esto tiene class o clases dependiendo de la ejecución de un método</p>
```

En este caso, cada vez que cambie cualquier cosa del estado del componente, se ejecutará de nuevo `obtenClases()` para recibir el nuevo valor de clases CSS que se deben aplicar al elemento.

Nota: El código de tu método `obtenClases()` debería ser muy conciso y rápido de ejecutar, puesto que su invocación se realizará muy repetidas veces. En definitiva, debes usarlo con cuidado para no afectar al rendimiento de la aplicación.

Evitar efectos laterales

Del mismo modo que ocurría con las expresiones en la Interpolación de strings, se debe evitar que al evaluar el valor de una propiedad se cambie el estado del componente.

Por ejemplo, si en la expresión a evaluar para definir el valor de una propiedad colocas una asignación, verás un error de Angular advirtiéndote este asunto.

```
<img [src]="ruta = 'ruta.jpg'">
```

Esto último no se puede hacer. Angular se quejará con un mensaje como "emplate parse errors: Parser Error: Bindings cannot contain assignments".

Ten en cuenta que, al bindear una propiedad mediante el valor devuelto por un método, Angular no mostrará errores frente a efectos laterales producidos por modificar el estado del componente. Por ejemplo en:

```
<img [src]="calculaSrc()">
```

Si en `calculaSrc()` cometemos la imprudencia de modificar el estado, Angular no será capaz de detectarlo y podremos encontrar inconsistencias entre las propiedades del componente y lo que se está mostrando en el template.

De momento es todo lo que tienes que saber sobre el bindeo de propiedades. En el siguiente artículo revisaremos otra de las preguntas frecuentes de las personas que comienzan con Angular, ¿Cuándo usar interpolación o bindeo a propiedades?.

Este artículo es obra de *Miguel Angel Alvarez*
Fue publicado / actualizado en 09/11/2017
Disponible online en <https://desarrolloweb.com/articulos/binding-propiedades-angular.html>

Binding a propiedad vs interpolación de strings, en Angular

Cuándo usar el binding de propiedad o la interpolación de strings, dos alternativas con objetivos distintos en Angular, que a veces pueden confundir.

En este artículo vamos a analizar en contraposición dos alternativas de binding en Angular, que a veces pueden producir dudas, básicamente porque podrían ser usadas en contextos similares.

El objetivo es ayudar al lector a identificar los casos donde se recomienda usar el bindeo de propiedades o la interpolación de strings, de modo que sea capaz de escoger la mejor opción en cada caso.

Recuerda que estos conceptos ya se han explicado, de modo que deberías conocer al menos qué diferencias tienen. Si no es así te recomendamos leer el artículo sobre la [interpolación de strings](#) y el [binding de propiedades](#).



Dos mecanismos distintos, que pueden usarse en situaciones parecidas

Sabiendo que la interpolación de strings y el binding de propiedades son cosas bien distintas, queremos mostrar que en ocasiones pueden ser usadas en los mismos lugares, produciendo idéntico resultado.

Observemos el siguiente código.

```
<p>
  
</p>
<p>
  <img [src]="urlImagen">
</p>
```

En el primer caso se está realizando interpolación, colocando como valor de una propiedad algo que viene de una variable del componente. En el segundo caso se está usando binding a la propiedad src, colocando el valor de la misma variable.

Ambos casos resuelven el HTML del componente con el mismo resultado. ¿Cuál es el correcto?

Tenemos más ejemplos. Observa ahora este otro caso, en el que básicamente colocamos una clase obtenida por medio de una propiedad del componente. El resultado será el mismo.

```
<ul>
  <li [class]="valorClass">item 1</li>
  <li class="{{valorClass}}">item 2</li>
</ul>
```

Por último veamos un caso donde también obtenemos el mismo resultado, pero usando dos aproximaciones mucho más diferentes.

```
<p>{{texto}}</p>
<p [innerHTML]="texto"></p>
```

En el primer párrafo estamos colocando como contenido un texto por interpolación. Mientras que en el segundo párrafo se coloca el mismo texto, a través de la propiedad del componente innerHTML.

Si estás confuso con ambas posibilidades sería más que normal. Intentaremos arrojar algo de luz sobre cuál de ellas te conviene usar en cada caso.

Uso de la interpolación con cadenas

En todos los casos anteriores estamos en definitiva volcando cadenas en diversos puntos del template. Aunque esas cadenas a veces se coloquen en propiedades del componente, no dejan de ser cadenas.

En todos los lugares donde se trate de colocar cadenas en una vista, puede que tenga más sentido usar interpolación, debido en mi opinión a la claridad del código de la vista.

Pero ojo, porque esto es una opinión personal, que no atiende a ningún motivo relacionado con el propio framework. De hecho, lo cierto es que no existe una diferencia real entre una u otra posibilidad, por lo que se pueden usar indistintamente.

Lo importante sería establecer una norma a seguir por todos los desarrolladores en cuanto a cuál usar en los casos en los que no exista diferencia, de modo que todo el mundo tome las mismas decisiones y obtengamos un código más homogéneo.

Uso de property binding con valores distintos de cadenas

El caso de valores que no sean cadenas de caracteres es ya distinto. Realmente la interpolación de strings siempre debe devolver una cadena, por lo que no será adecuada si aquello que se

tiene que usar en una propiedad no es directamente una cadena.

Por ejemplo, la propiedad "disabled" de un botón debe igualarse por property binding a un valor boleano, luego no será adecuado usar interpolación. Por tanto, lo correcto sería esto:

```
<button [disabled]="activo">Clic aquí</button>
```

En este caso concreto, no deberíamos usar interpolación, ya que el resultado no será el deseado. Por ejemplo, esto no sería correcto.

```
<button disabled="{{activo}}">Clic aquí</button>
```

El motivo es que activo se interpolaría por la palabra "false". En este caso disabled="false" en HTML sería lo mismo que colocar el atributo "disabled" sin acompañarlo de ningún valor, lo que equivale al final a que el botón esté desactivado.

En definitiva, cada vez que tengamos que usar un dato que no sea una cadena, es preferible colocar el binding de propiedad. Por ejemplo, la directiva ngStyle espera un objeto como valor, por lo que estaríamos obligados a usar property binding.

```
<div [ngStyle]="estilo">Test de estilo</div>
```

Sanitización de cadenas en templates de Angular

En el caso que tanto property binding como string interpolation se usen para volcar cadenas en el template, hay una diferencia destacable en la manera como las cadenas se van a sanitizar antes de volcarse al template.

En vista de esta propiedad en el componente:

```
cadenaXSS = 'Esto es un <script>alert("test")</script> con XSS';
```

La propiedad cadenaXSS presenta un problema de seguridad, por una inyección de código de un script, lo que se conoce por XSS.

Podríamos volcarla en un template de la siguiente manera:

```
<div>{{ cadenaXSS }}</div>  
<div [innerHTML]="cadenaXSS"></div>
```

Angular en ambos casos hace un saneamiento de la cadena, desactivando la etiqueta SCRIPT, para evitar problemas de seguridad por inyección de código XSS. Pero el resultado de ese saneamiento es diferente en cada caso.

En este caso concreto, la salida que obtendremos es la siguiente:

Esto es un `<script>alert("test")</script>` con XSS Esto es un `alert("test")` con XSS

Tienes que observar que, en el caso de la interpolación de strings, la etiqueta script te aparece como texto en la página. Es decir, será un simple texto y no una apertura y cierre de una etiqueta para colocar un script Javascript.

Esperamos que con estas indicaciones hayas resuelto algunas dudas típicas que pueden surgir cuando empiezas a trabajar en el desarrollo con Angular. Ahora estás en condiciones de escoger la mejor opción en cada caso, entre string interpolation y Property binding.

Este artículo es obra de *Miguel Angel Alvarez*
Fue publicado / actualizado en 23/11/2017
Disponible online en <https://desarrolloweb.com/articulos/binding-propiedad-vs-interpolacion-strings.html>

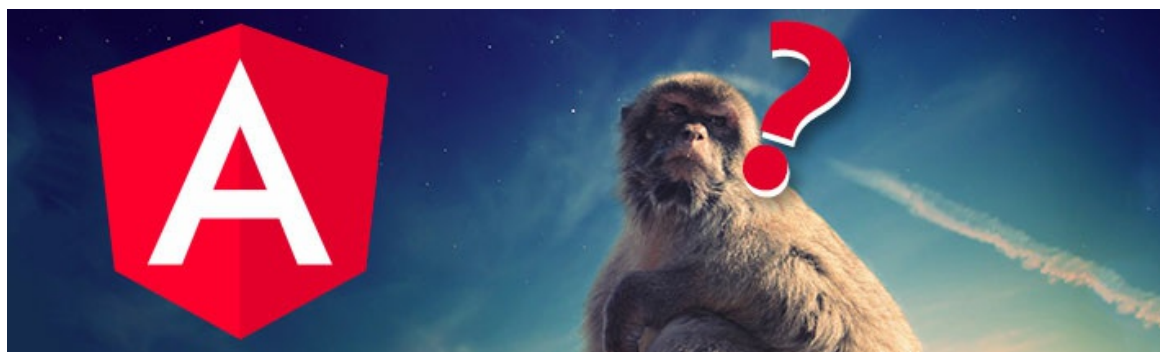
Binding a atributo en Angular

Qué es el binding a atributo, por qué debemos conocerlo. En qué casos aplica el binding de atributo y ejemplos de uso.

En este artículo vamos a conocer una cosa nueva en el Manual de Angular. Está relacionada con el binding, que hemos tratado en numerosas ocasiones, aunque se trata de un nuevo caso de binding del que no habíamos hablado anteriormente: el binding a atributo.

En si, usar el binding a atributo es muy sencillo, pues resulta en una sintaxis muy elemental que tenemos que usar. Lo complicado puede ser ver el concepto y saber los momentos en los que se debe usar. Desde luego son pocos, pero si no conocemos este concepto nos encontraremos a veces con errores de Angular que nos puede costar solucionar.

Comenzaremos por explicar qué se conoce como atributo, frente a una propiedad, y luego explicaremos la sintaxis que usaremos cuando necesitemos bindear a un atributo.



Qué es un atributo en Angular

Primero cabe decir que en este caso vamos a definir "atributo" en el contexto de Angular, ya

que en el contexto de HTML entendemos atributo como los modificadores que acompañan a etiquetas.

```
<p class="miClase">En este párrafo usamos el atributo "class"</p>
```

El HTML tiene cientos de atributos y debes saber perfectamente cómo funcionan, pero no nos referimos a ellos, sino a atributos en el contexto del desarrollo en Angular.

Incluso en Angular, a menudo usamos la palabra "atributo" como sinónimo de "propiedad". En muchos casos podría valer meter todo en el mismo saco, pero lo cierto es que no siempre es lo mismo y Angular no trata a ambos de la misma manera.

Básicamente, propiedad se refiere a una característica de un objeto del DOM, definida por una propiedad del objeto de ese elemento del DOM. En muchas ocasiones, cuando escribes HTML existe una correspondencia directa entre lo que sería un atributo de HTML y lo que sería una propiedad del DOM.

Por ejemplo el atributo "id" del HTML corresponde directamente con la propiedad "id" del objeto del DOM. Pero existen atributos que no tienen un mapeo directo en el DOM, como es el caso de "colspan". Existen atributos de HTML que mapean a propiedades del DOM que no se llaman ni funcionan igual, como "class" y "classList". En general esta es la clave de la cuestión.

Nota: Sin entrar en Angular, atributos en el HTML que tienen correspondencia con el DOM, como "id" o "value". Sin embargo hay una diferencia fundamental en cuanto al funcionamiento de un atributo y de una propiedad. En el HTML los atributos sirven para inicializar propiedades del DOM, sin embargo, durante la vida del elemento la propiedad puede cambiar (escribiendo otra cosa en el campo de texto cambia su propiedad "value") pero el atributo permanece exactamente igual. El atributo se utilizó para inicializar la propiedad del objeto, pero luego aunque ese objeto cambie, el atributo en el HTML sigue igual. Pruébalo tú mismo. Coloca un campo de texto con un value="xx". Luego edita el campo de texto, colocando cualquier otra cadena. En el momento que lo editas cambia la propiedad "value" del DOM. Pero luego intenta hacer sobre el campo de texto "elementoInputDelDom.getAttribute('value')" y verás que el valor del atributo que te devuelve continua siendo "xx".

Pues bien, cuando hacemos bindeo a propiedades con Angular (property binding), tenemos disponibles las propiedades del elemento, pero no los atributos. Por tanto, podemos hacer esto:

```
<div [id]="idDivision">Test de binding a propiedad</div>
```

Pero no podemos hacer esto otro:

```
<td [colspan]="1 + 1">test</td>
```

En el momento en el que intentamos hacer un binding a propiedad, sobre una propiedad que

no existe en el DOM del elemento y que Angular no entiende, obtendremos un mensaje como este: Can't bind to [...] since it isn't a known property of [...].

```
✖ Uncaught Error: Template parse errors: compiler.es5.js:1694
  Can't bind to 'colspan' since it isn't a known property of 'td'. ("
  <table>
    <td [ERROR ->][colspan]="1 + 1">test</td>
  </table>
```

Ni tan siquiera podemos acudir a la interpolación de strings para resolver este problema. Si intentamos algo como esto tampoco funcionará:

```
<td colspan="{{1 + 1}}">test</td>
```

El asunto es que, tanto string interpolation como property binding son capaces de bindear únicamente a propiedades definidas en Angular y no a atributos del HTML.

Sintaxis para bindear a un atributo

La sintaxis necesaria para que Angular procese el binding a un atributo, sin producir ningún mensaje de error, es muy parecida a la sintaxis de binding a una propiedad que ya conocemos. Solo tenemos que especificar que en ese preciso momento estamos bindeando a un atributo. Si antes colocábamos la propiedad entre corchetes, ahora colocaremos "attr." seguido del nombre del atributo.

```
<td [attr.colspan]="1 + 1">test</td>
```

Por ver otro ejemplo, los "data attribute" tampoco tienen mapeo a propiedades. En estos casos, si queremos bindear a un "data attribute" del HTML5, tendremos que hacerlo con la sintaxis que acabamos de conocer.

Por si no sabes qué es un "data attribute", tiene un aspecto como este:

```
<div data-cliente="DesarrolloWeb.com">Desarrollo Web</div>
```

Ahora, si en el TypeScript de nuestro componente tenemos un objeto cliente definido, como podría ser:

```
cliente = {
  nombre: 'DesarrolloWeb.com'
}
```

Y queremos bindear ese nombre en el data attribute, haríamos algo como esto

```
<div [attr.data-cliente]="cliente.nombre">{{cliente.nombre}}</div>
```

Ejemplo completo de componente que implementa bindeo a atributo

Vamos a ver ahora un ejemplo de un componente que realiza el binding a atributo, de modo que podamos practicar algo. Es un componente que nos muestra los puntos donde se situa un polígono. Queremos mostrar los datos del polígono haciendo uso de una tabla en la que hay una casilla que está expandida en varias columnas.

La tabla resultante que queremos conseguir tendrá este aspecto:

Nombre	Lados	Puntos			
cuadrado	4	10,5	15,5	15,10	10,10

PONER IMAGEN

Comenzaremos viendo la definición TypeScript de nuestro componente, en la que podremos observar la especificación de los datos de nuestro polígono en una propiedad "poligono" del componente.

```
import { Component, OnInit } from '@angular/core';

@Component({
  selector: 'dw-datos-poligono',
  templateUrl: './datos-poligono.component.html',
  styleUrls: ['./datos-poligono.component.css']
})
export class DatosPoligonoComponent implements OnInit {

  poligono = {
    nombre: 'cuadrado',
    puntos: [
      {
        x: 10,
        y: 5
      },
      {
        x: 15,
        y: 5
      },
      {
        x: 15,
        y: 10
      },
      {
        x: 10,
        y: 10
      }
    ]
  }

  constructor() {}

  ngOnInit() {}
}
```

Ahora veamos la parte más representativa de este componente, en el objetivo del artículo, de aprender el bindeo a atributos.

```

<table>
<tr>
<th>Nombre</th>
<th>Lados</th>
<th [attr.colspan]="poligono.puntos.length">Puntos</th>
</tr>
<tr>
<td>{{poligono.nombre}}</td>
<td>{{poligono.puntos.length}}</td>
<td *ngFor="let punto of poligono.puntos">{{punto.x}},{{punto.y}}</td>
</tr>
</table>

```

El bindeo de atributo nos permite hacer que la celda de la tabla donde pone "puntos" se expanda en un número de celdas igual al número de puntos del polígono. Dependiendo de la definición de ese polígono las celdas de la tabla se adaptarán a las necesidades de cada caso.

Luego nos quedaría simplemente colocar unos estilos, a placer. Que, para que se parezcan a la tabla que hemos mostrado en la imagen de antes, podrían ser como estos:

```

table {
border: 1px solid darkgoldenrod;
margin: 20px;
}
th {
background-color: rgb(220, 224, 152);
text-align: center;
padding: 10px;
}
td {
padding: 5px;
border: 1px solid #ddd;
}

```

Eso es todo. Como has visto, el bindeo a atributos es muy sencillo. Sólo hay que tener claro el concepto y cuándo se debe aplicar. Ahora si te encuentras en una situación similar, simplemente acuérdate de este artículo, o busca en Google ;).

Este artículo es obra de *Miguel Angel Alvarez*
Fue publicado / actualizado en 27/02/2018
Disponible online en <https://desarrolloweb.com/articulos/binding-atributo-angular.html>

Declaraciones de eventos en templates Angular al detalle

Qué son template statements y cómo declarar comportamientos ante eventos en Angular. Contexto del template accesible en las sentencias de eventos.

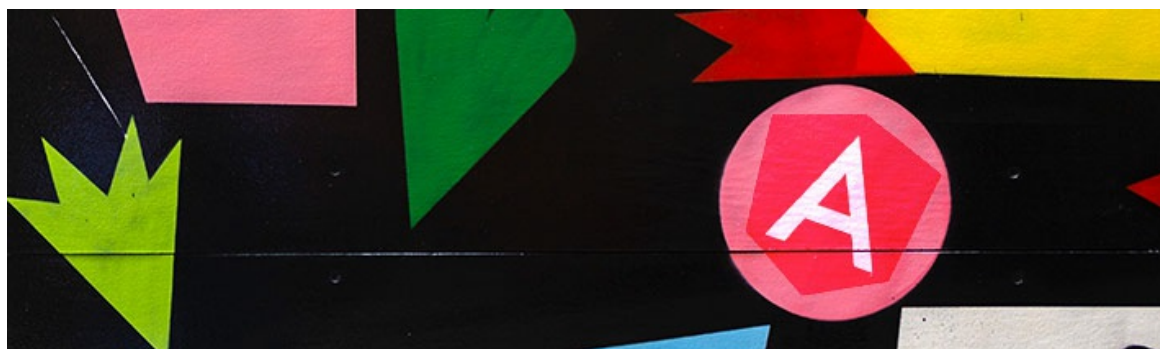
En este artículo vamos a ofrecer un poco más de información sobre cosas que puedes hacer cuando estás trabajando con eventos en componentes de Angular. Son informaciones que quizás conozcas, pues las hemos abordado a través de diversas prácticas en lo que llevamos de Manual de Angular., aunque queremos llegar un poco más lejos que antes y ofrecer datos

adicionales que te ayuden a usar el framework con mayor soltura.

La sintaxis para declarar un código a ejecutar como respuesta a un evento ya fue tratada en la introducción de este manual, en el artículo [sintaxis para las vistas de componentes](#). No obstante, recordemos lo visto a través del siguiente código.

```
<button (click)="procesarClic()">Haz clic!</button>
```

El evento en cuestión se coloca entre paréntesis y luego hay que asignarle una sentencia a ejecutar como respuesta a ese evento.



Template statements

La sentencia a ejecutar como respuesta a un evento se conoce con el nombre de "statement" en la documentación de Angular. Un statement es la declaración del comportamiento que debe realizarse al dispararse un evento.

Es habitual que un statement produzca la ejecución de un método, como en el BUTTON anterior. Eso permite que mantengamos el código de la lógica del programa en el TypeScript, lo que en principio es más lógico, pero también podríamos contener directamente la sentencia en el propio template.

```
<div (click)="cliente.nombre = 'DesarrolloWeb.com'">Marcar Cliente</div>
```

En cualquier caso, un statement es simplemente la declaración de un código, en el que se usa una sintaxis similar a Javascript (o realmente similar a TypeScript). El statement se declara entre comillas, como se puede ver en el código anterior.

Los statements están pensados para modificar el estado

Así como las expresiones nunca debían modificar el estado de la aplicación, estando prohibidas sentencias como la asignación, los statements hacen el camino contrario. Dicho de otro modo, los statements están pensados para alterar las propiedades del componente, produciendo un método de comunicación desde las vistas (template) hacia su lógica.

Por este motivo, es habitual que en un statement se usen operadores de asignación, para

modificar las propiedades de un componente. Incluso pueden encadenarse varias sentencias si se desea, usando el ";" para separarlas.

Sin embargo, hay algunas cosas que no están permitidas en las sentencias usadas para ejecutar un evento, como por ejemplo.

- El operador new
- Operadores de incremento y decremento (++ y --)
- Asignaciones con operaciones en un mismo paso (+= y similares)
- Pipes

El contexto de los statements

De modo que en las expresiones, los statements tienen disponible el contexto del componente, pudiendo acceder a las propiedades como si fueran variables, o a sus métodos.

Esto ya lo conocemos porque lo hemos usado varias veces, aunque en el caso concreto de los statements hay algunas cosas interesantes que cabe destacar.

Objeto \$event

Este objeto es interesante porque nos ofrece información sobre el evento que se acaba de producir.

```
<button (click)="procesarEvento($event)">Transmito el objeto evento</button>
```

En el anterior ejemplo, al invocar al método procesarEvento() se le está enviando \$event, que contiene información sobre el evento click que se realizó.

Nota: \$event es muy interesante, porque nos permitirá también enviar datos útiles cuando comencemos a utilizar los eventos personalizados en componentes.

Variables locales generadas en estructuras de template

En los templates de repetición se genera un contexto adicional, al que también se puede acceder por medio de los statements. Si recordamos el código de un *ngFor <https://www.desarrolloweb.com/articulos/directiva-ngfor-angular2.html> podremos saber a lo que nos referimos.

```
<article
  *ngFor="let contactItem of contactos"
  (click)="seleccionaContacto(contactItem)"
>
```

```
{{ contactoItem }}  
</article>
```

Como puedes ver, en la repetición producida por `*ngFor`, se genera un nuevo contexto con la variable "contactoItem", que contiene el elemento actual de la repetición. Esa variable está disponible en el statement y la podemos enviar al método `seleccionaContacto()` para su utilización. En el método recibimos esa variable como un parámetro.

```
seleccionaContacto(contacto) {  
  console.log('Se ha hecho clic sobre el contacto', contacto);  
}
```

Otro estilo de variable generada en un template es la referencia a un elemento. Por ejemplo en este código tenemos un párrafo en el que hemos definido una referencia "#parrafoTest".

```
<p #parrafoTest (click)="procesaClic(parrafoTest)">Test</p>
```

Esa referencia la podemos usar para enviarla a un método. Tal referencia no es más que el elemento en cuestión, el propio elemento del DOM. Podríamos usar ese elemento para ejecutar cualquier código con él.

```
procesaClic(p) {  
  p.style.color = 'red';  
}
```

En el pasado método recibimos el párrafo y le asignamos un color rojo.

Consejos a la hora de usar statements

Un par de consejos adicionales para producir sentencias en aplicaciones fácilmente mantenibles son los siguientes.

- Es preferible definir el statement mediante un método del componente, de modo que la funcionalidad y la lógica del componente se quede en el código TypeScript.
- Si optas por colocar el código a ejecutar en el statement (no usar un método del componente) debería limitarse a una simple asignación.

Conclusión

La declaración de eventos es algo básico en Angular, pero no la habíamos tratado hasta ahora con todo el detalle que merecía. Esperamos que, aunque ya estuvieses trabajando con tus propios eventos en componentes, este artículo te haya enseñado cosas nuevas.

En la sección dedicada a binding del [Manual de Angular](http://desarrolloweb.com/manuales/manual-angular-2.html) hemos conocido todas las cosas básicas que podemos hacer con el framework, aunque todavía no hemos abordado la interacción entre componentes. Para ello en los próximos artículos nos dedicaremos a conocer `@Input` y `@Output`.

Este artículo es obra de *Miguel Angel Alvarez*
Fue publicado / actualizado en 13/03/2018
Disponible online en <https://desarrolloweb.com/articulos/declaraciones-eventos-statements-angular.html>

Comunicación entre componentes con @Input

Aprende a comunicar entre los componentes, enviando propiedades desde el componente padre al hijo. Para ello usaremos el decorador @Input de Angular, que define que ciertas propiedades del componente puedan asignarse desde el template padre.



En artículos anteriores del Manual de Angular, al abordar la introducción al framework, hemos mencionado en varias ocasiones una de sus características fundamentales: la arquitectura de componentes.

Hemos dicho que una aplicación en Angular la forma un árbol de componentes, capaces de colaborar entre sí para resolver las necesidades del negocio. Pero lo cierto es que, hasta el momento nuestros componentes han funcionado de manera bastante autónoma, sin la posibilidad de interaccionar los unos con los otros.

En este artículo vamos a comenzar con la revisión de los procesos de comunicación habituales entre los componentes de aplicaciones, analizando especialmente el uso de las propiedades de entrada decoradas con @Input().

Envío de datos entre componentes

Es verdad que en pasados artículos hemos visto como un servicio es capaz de facilitar que los componentes compartan datos, pero no es la única manera que podemos conseguir ese objetivo.

Además de los servicios como comunicadores de datos, existen diferentes escenarios mediante los cuales los componentes pueden compartir información. Atendiendo a la estructura de árbol de componentes en la aplicación, esa comunicación puede surgir de los padres a los hijos y de los hijos hacia los padres.

Paso de datos de padres a hijos

El paso de información de padres a hijos se realiza por medio de propiedades del componente. La información se bindea desde el template, usando los atributos de la etiqueta del componente.

Para ello, las propiedades del componente se deben decorar con `@Input`, de modo que Angular sea capaz de saber que éstas son capaces de inicializarse, o modificarse, desde fuera. Este es el escenario que vamos a abordar en este artículo.

Paso de datos de los hijos a los padres

También puede surgir la necesidad de que los hijos comuniquen a los padres cambios en la información que ellos manejan. En estos casos Angular utiliza eventos. Es decir, cuando el hijo tiene un dato y quiere hacerlo llegar al padre, genera un evento que puede ser capturado en el padre para realizar aquellas acciones que sean necesarias.

Para definir los eventos que van de los hijos a los padres, Angular usa el decorador `@Output`, que veremos con detalle en el siguiente artículo.

Nota: El viaje de los datos de los padres a los hijos por medio de sus propiedades y la escalación de eventos para comunicar los datos de hijos a padres tiene el nombre de patrón "mediator".

Paso de datos desde componentes padre a componentes hijo

Ahora vamos a ver cómo se realiza el envío de datos del padre al hijo, comenzando por la definición del dato en el template.

Vamos a suponer que tenemos un componente que se llama `VisualizarClienteComponent`. Este componente muestra la ficha de un cliente, con sus datos principales (nombre, CIF...). Para funcionar, este componente debe recibir del padre los datos del cliente que debe mostrar.

Por otra parte, tenemos otro componente que usa a `VisualizarClienteComponent`, que le tiene que pasar los datos que debe mostrar. Al usar el componente, tiene que pasar los datos por medio del template, usando el binding de propiedades.

El uso del componente sería como este:

```
<dw-visualizar-cliente
  [nombre]="cliente.nombre"
  [cif]="cliente.cif"
  [direccion]="cliente.direccion"
></dw-visualizar-cliente>
```

Como puedes ver, las propiedades se pasan por medio de binding, usando la sintaxis de los corchetes.

A la vista de este código se entiende que el componente padre tendrá un objeto cliente. El componente hijo tendrá tres propiedades, "nombre", "cif" y "direccion". A la propiedad nombre le pasaremos el valor de "cliente.nombre", a la propiedad "cif" le pasaremos el valor de "cliente.cif", etc.

Nota: nada impide haber pasado el objeto entero, de una sola vez en una única propiedad, en lugar de sus valores por separado como en el ejemplo. Sería incluso más claro y hasta preferible en muchos casos en una aplicación. Sin embargo, a fines didácticos he preferido separar los datos en varias propiedades, por el motivo de poder expresar con más claridad qué pertenece a cada componente, entre hijo y padre.

Declaración de propiedades de entrada con @Input

A continuación vamos a ver cómo definir en el componente hijo las propiedades que son capaces de cargar datos desde el padre. Para ello usaremos el decorador @Input.

Importar el decorador @Input

Como un primer paso en el componente hijo necesitamos hacer el import del decorador Input, que está en "@angular/core".

```
import { Component, OnInit, Input } from '@angular/core';
```

Declarar las propiedades con su decorador @Input

También en el código TypeScript del componente hijo, tenemos que declarar las propiedades en la clase, igual que veníamos haciendo, solo que agregaremos el decorador @Input para indicar que esta propiedad se puede manipular desde el padre.

De momento vamos a dejar el decorador vacío, que será suficiente en la mayoría de los casos, aunque podríamos indicar algunas cuestiones como el nombre con el que conocerá la propiedad desde fuera.

```
@Input()  
nombre: string;
```

Si se desea, es posible aplicar un valor predeterminado a la propiedad, asignando un valor cualquiera. En ese caso, si no se entrega ese valor al usar el componente, se asumirá el definido de manera predeterminada.

```
@Input()  
nombre = 'DesarrolloWeb.com';
```

Nota: En este caso no es necesario que indiques que el tipo de la propiedad es "string", ya que el compilador de TypeScript es capaz de inferirlo en función del dato asignado.

Código completo del componente

Aparte de los `@Input`, el código del componente no tiene nada que no se conozca hasta el momento. No obstante dejamos el código completo a continuación.

```
import { Component, OnInit, Input } from '@angular/core';

@Component({
  selector: 'dw-visualizar-cliente',
  templateUrl: './visualizar-cliente.component.html',
  styleUrls: ['./visualizar-cliente.component.css']
})
export class VisualizarClienteComponent implements OnInit {

  @Input()
  nombre = 'DesarrolloWeb.com';
  @Input()
  cif: string;
  @Input()
  direccion: string;

  constructor() { }

  ngOnInit() {
  }

}
```

El template del componente tampoco tiene ningún detalle a resaltar, puesto que las propiedades se usan igual que hemos visto en anteriores ocasiones.

```
<p>
  Nombre: {{nombre}}
<br>
  Cif: {{cif}}
<br>
  direccion: {{direccion}}
</p>
```

Binding de una dirección

Estos datos, tal como se ha dicho, viajan del padre al hijo. Si el padre modifica el valor enviado a las propiedades, también se modificará en el hijo. Lo puedes probar si en el template donde has usado el componente pones un botón para que modifique una de sus propiedades.

```
<button (click)="cliente.cif='ESB9922'">Cambiar CIF</button>
```

Al pulsar el botón se modificará el CIF y por tanto cambiará el componente que consumía este dato.

Recuerda que el binding es de una sola dirección, como todos los binding de propiedad expresados con los corchetes. Por tanto, si el hijo modifica la información de la propiedad, el nuevo valor no viajará hacia el padre.

En el siguiente artículo te mostraremos [cómo hacer que un dato pueda viajar desde el hijo al padre, usando @Output](#).

Este artículo es obra de *Miguel Angel Alvarez*
Fue publicado / actualizado en 13/05/2020
Disponible online en <https://desarrolloweb.com/articulos/comunicacion-componentes-input.html>

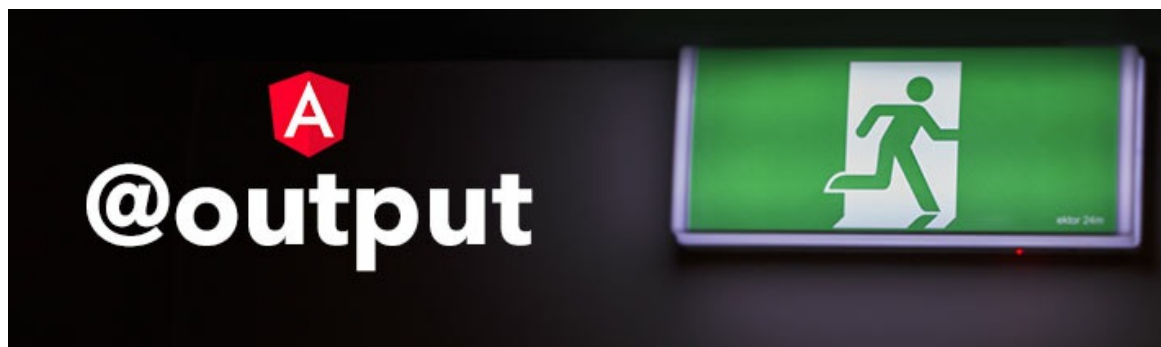
Emisión de eventos personalizados con @Output en Angular

Explicamos la comunicación de cambios de los hijos hacia los padres por medio de eventos personalizados, generados con propiedades @Output.

En el pasado artículo del [Manual de Angular](#) conocimos las [propiedades @Input](#), que permitían comunicar datos desde el componente padre hacia el componente hijo.

También comentamos que existe la posibilidad de implementar la otra dirección en la comunicación entre componentes: desde los hijos hacia los padres. Ésta se realiza mediante la emisión de un evento personalizado, pero no llegamos a explicar cómo se implementaba.

Así que vamos a poner manos a la obra para conocer las propiedades @Output, que nos permitirán la emisión de esos eventos personalizados, con los que avisar a los padres de cualquier situación que ocurra en los hijos y que deban conocer.



Para aclarar las ideas, resumimos el esquema de trabajo de la comunicación de hijos a padres, en el que están involucrados dos componentes:

- El componente hijo será el encargado de escalar el evento hacia el padre, para avisarle de un suceso. Al avisarle, el hijo podrá comunicar un dato que el padre deba conocer, relacionado lógicamente con ese suceso.
- El componente padre será capaz de capturar el evento personalizado emitido por el hijo y recuperar aquel dato que fue enviado.

Implementar el evento personalizado en el componente hijo

Nuestro trabajo en la comunicación de hijos a padres comienza en el componente hijo. El proceso de trabajo no es trivial y su realización implica el uso de varios actores. Para explicarlo comenzaremos presentando las partes implicadas en el proceso.

Clase EventEmitter

La clase de Angular que implementa objetos capaces de emitir un evento se llama "EventEmitter". Pertenecce al "core" de Angular, por lo que necesitamos asegurarnos de importarla debidamente.

```
import { Component, OnInit, EventEmitter } from '@angular/core';
```

Para poder emitir eventos personalizados necesitaremos crear una propiedad en el componente, donde instanciar un objeto de esta clase (new EventEmitter). Solo que este proceso tiene un detalle que lo puede complicar un poco, al menos al principio, ya que hace uso de los "generics" de TypeScript.

Básicamente, el genérico nos sirve para decirle a TypeScript el tipo del dato que nuestro evento personalizado escalará hacia el padre en su comunicación. Este es el código que podríamos utilizar para crear nuestra propiedad emisora de eventos, haciendo uso del generics.

```
propagar = new EventEmitter<string>();
```

Así le estamos diciendo que nuestro emisor de eventos será capaz de emitir datos de tipo "string". Obviamente, este tipo de dato dependerá del componente y podrá ser no solamente un tipo primitivo, sino una clase o una interfaz.

Nota: Si no entiendes mucho esta parte, te recomendamos leer este artículo con mayor información de los [genéricos de TypeScript](#).

El nombre de la propiedad, "propagar" en este caso, es importante, porque a la hora de capturar el evento en el padre tendremos que usarlo tal cual.

Decorador @Output

La propiedad de tipo "EventEmitter", necesaria para emitir el evento personalizado, debe ser decorada con @Output. Esto le dice al framework que va a existir una vía de comunicación desde el hijo al padre.

Para usar ese decorador tenemos que importarlo también. Igual que EventEmitter, la declaración de la función decoradora "Output" está dentro de "@angular/core". Nuestro

import, quedará por tanto así.

```
import { Component, OnInit, Output, EventEmitter } from '@angular/core';
```

Para usar el decorador colocaremos la llamada con los paréntesis vacíos, lo que será suficiente en la mayoría de los casos. El código de la declaración de la propiedad, usando el correspondiente decorador `@Output()` es el siguiente:

```
@Output()
propagar = new EventEmitter<string>();
```

Método emit()

Cuando queramos disparar el evento personalizado invocaremos el método `emit()` del objeto `EventEmitter`.

Este método recibe como parámetro aquel dato que queramos hacer llegar al padre. Lógicamente, el tipo del dato que enviemos hacia el padre debe concordar con el que hayamos declarado en el genérico al instanciar la el objeto `EventEmitter`.

```
this.propagar.emit('Este dato viajará hacia el padre');
```

Ejemplo de componente que envía eventos al padre

Con lo que sabemos ahora ya somos capaces de enfrentarnos a nuestro primer ejemplo de componente con propiedades `@Output`.

Vamos a realizar un sencillo componente que tiene un campo de texto y un botón. Al pulsar el botón emitirá un evento hacia el padre, pasando la cadena escrita en el campo de texto.

Comenzaré mostrando el template del componente, que es la parte que nos ayudará a entender el comportamiento del componente.

```
<p>
  <input type="text" [(ngModel)]="mensaje">
  <button (click)="onPropagar()">Propagar</button>
</p>
```

Como puedes ver, tenemos el campo de texto, cuyo texto escrito está bindeado a la propiedad "mensaje".

Luego tenemos un botón que al hacer clic sobre él, ejecutará el método "onPropagar()".

Código TypeScript del componente

A continuación encontramos el código TypeScript del componente en cuestión. Vamos a verlo todo de una vez y luego explicaremos algunos detalles, aunque con los conceptos ya tratados anteriormente debería estar más o menos claro.

```
import { Component, OnInit, Output, EventEmitter } from '@angular/core';

@Component({
  selector: 'dw-propagador',
  templateUrl: './propagador.component.html',
  styleUrls: ['./propagador.component.css']
})
export class PropagadorComponent implements OnInit {

  mensaje: string;

  @Output()
  propagar = new EventEmitter<string>();

  constructor() {}

  ngOnInit() {}

  onPropagar() {
    this.propagar.emit(this.mensaje);
  }
}
```

Debes fijarte en los siguientes puntos.

- Hacemos el import de "Output" y "EventEmitter".
- Creamos la propiedad "mensaje", de tipo string. Esta propiedad contiene el texto que vamos a enviar hacia el padre cuando se dispare el evento personalizado.
- Creamos la propiedad "propagar". Esta es la más importante del presente ejemplo, pues es el emisor del evento. Observa que usamos el decorador @Output y que se declara como un objeto de la clase "EventEmitter". Además estamos indicando en el genérico de TypeScript que el tipo del dato que escalará hacia el padre es un string.
- Tenemos el método onPropagar(), que se invoca al hacer clic en el botón. Ese método se encarga de emitir el evento personalizado, con this.propagar.emit(), que viajará hacia el padre. En la emisión de ese evento será entregado al padre el dato enviado por parámetro a emit(), que es el valor de la propiedad, de tipo string, llamada "mensaje".

Con esto hemos acabado nuestro trabajo con el componente hijo. Ahora nos queda usar ese componente y recibir los eventos.

Recibir eventos personalizados en el componente padre

Ahora nos vamos a centrar en el componente padre. Es el que recibirá los eventos emitidos por el hijo. Es una operativa bastante sencilla, ya que los eventos personalizados se reciben igual que los eventos que genera el navegador.

Al usar el componente hijo debemos especificar el correspondiente manejador de evento en el template del padre. El nombre del evento personalizado será el mismo con el que se definió anteriormente la propiedad emisora de evento en el hijo.

```
<dw-propagador  
  (propagar)="procesaPropagar($event)"  
></dw-propagador>
```

Como recordarás, los manejadores de eventos se declaran con el nombre del evento entre paréntesis. Como decíamos, el nombre del evento corresponde con el nombre de la propiedad en el componente hijo.

El manejador del evento se indica entre las comillas dobles. Ese manejador "procesaPropagar" es un método en el componente padre. Es interesante aquí el uso del parámetro "\$event", que contiene el dato que está viajando del hijo al padre.

Ahora podemos ver el código de "procesaPropagar", que realizará las acciones necesarias para procesar el evento.

```
procesaPropagar(mensaje) {  
  console.log(mensaje);  
}
```

En nuestro caso solo estamos mostrando en la consola el dato "mensaje" que contiene la cadena enviada del hijo al padre.

Eso es todo! hemos completado el proceso de comunicación de hijos a padres, realizado mediante la emisión de eventos personalizados en Angular. Quizás pueda parecer un poco complicado en un principio, pero a fuerza de repetir lo encontrarás bastante elemental.

Este artículo es obra de *Miguel Angel Alvarez*
Fue publicado / actualizado en 18/04/2018
Disponible online en <https://desarrolloweb.com/articulos/emision-eventos-output-angular.html>

Servicios en Angular

Comenzamos una serie de artículos que nos explicarán qué son los servicios, una de las piezas principales de las aplicaciones en Angular. Estudiaremos cómo usar servicios dentro de aplicaciones, cómo crearlos, cómo agregarlos a módulos o componentes, cómo instanciar servicios, etc. Veremos también qué papel juegan los servicios dentro del marco de una aplicación, con ejemplos de uso que puedan ilustrar las buenas prácticas a los programadores.

Servicios en Angular

Primero de los artículos dedicado a los servicios de Angular. Veremos qué son los servicios en el framework Javascript Angular, cómo crear services e inyectarlos en los componentes para acceder a ellos.



Hasta ahora en el [Manual de Angular](#) hemos hablado mucho de componentes, y también de módulos. Pero existen otros tipos de artefactos que podemos usar para organizar el código de nuestras aplicaciones de una manera más lógica y fácil de mantener.

En importancia a componentes y módulos le siguen los servicios. Con este artículo iniciamos su descripción. Los servicios, o "services" si lo prefieres en inglés, son una de las piezas fundamentales en el desarrollo de aplicaciones Angular. Veremos qué es un servicio y cómo dar nuestros primeros pasos en su creación y utilización en un proyecto.

Concepto de servicio en Angular

Si eres ya viejo conocido del desarrollo con Angular sabrás lo que es un servicio, pues es una parte importante dentro de la arquitectura de las aplicaciones con este framework. Pero si eres nuevo seguro que te vendrá bien tener una pequeña descripción de lo que es un servicio.

Básicamente hemos dicho anteriormente que el protagonista en las aplicaciones de Angular es el componente, que las aplicaciones se desarrollan en base a un árbol de componentes. Sin embargo, a medida que nuestros objetivos sean más y más complejos, lo normal es que el código de los componentes también vaya aumentando, implementando mucha lógica del

modelo de negocio.

En principio esto no sería tan problemático, pero sabemos que la organización del código, manteniendo piezas pequeñas de responsabilidad reducida, es siempre muy positiva. Además, siempre llegará el momento en el que dos o más componentes tengan que acceder a los mismos datos y hacer operaciones similares con ellos, que podrían obligarnos a repetir código. Para solucionar estas situaciones tenemos a los servicios.

Básicamente un servicio es un proveedor de datos, que mantiene lógica de acceso a ellos y operativa relacionada con el negocio y las cosas que se hacen con los datos dentro de una aplicación. Los servicios serán consumidos por los componentes, que delegarán en ellos la responsabilidad de acceder a la información y la realización de operaciones con los datos.

Cómo crear un servicio

Tal como viene siendo costumbre en el desarrollo con Angular, nos apoyaremos en [Angular CLI](#) para la creación del esqueleto, o scaffolding, de un servicio.

Para crear un servicio usamos el comando "generate service", indicando a continuación el nombre del servicio que queremos generar.

```
ng generate service clientes
```

Esto nos generaría el servicio llamado "ClientesService". La coletilla "Service", al final del nombre, te la agrega Angular CLI, así como también nombra al archivo generado con la finalización "-service", para dejar bien claro que es un servicio.

Es habitual que quieras colocar el servicio dentro de un módulo en concreto, para lo que puedes indicar el nombre del módulo, una barra "/" y el nombre del servicio. Pero atención: ahora lo detallaremos mejor, pero queremos advertir ya que esto no agregará el servicio al código de un módulo concreto, sino que colocará el archivo en el directorio de ese módulo. Enseguida veremos qué tienes que hacer para que este servicio se asigne realmente al módulo que desees.

```
ng generate service facturacion/clientes
```

Nota: como los servicios son algo que se suele usar desde varios componentes, muchos desarrolladores optan por crearlos dentro de un módulo compartido, que puede llamarse "común", "shared" o algo parecido.

Ahora, si quieres, puedes echarle un vistazo al código básico de un service, generado por el CLI. Enseguida lo examinamos. No obstante, antes queremos explicar otro paso importante para que el servicio se pueda usar dentro de la aplicación.

Para poder usar este servicio es necesario que lo agregues a un módulo. Inmediatamente lo podrás usar en cualquiera de los componentes que pertenecen a este módulo. Este paso es importante que lo hagas, puesto que, al contrario de lo que ocurriría al crear un componente con el CLI, la creación de un servicio no incluye la modificación del módulo donde lo has creado.

Así pues, vamos a tener que declarar el servicio manualmente en el módulo. Lo haremos gracias al decorador del módulo (`@NgModule`), en el array de "providers". El decorador de un módulo con el array de providers podría quedar más o menos así.

```
@NgModule({
  imports: [
    CommonModule
  ],
  declarations: [ListadoClientesComponent],
  providers: [ClientesService]
})
```

Obviamente, tendrás que hacer el correspondiente import al módulo, para que se conozca la clase `ClientesService`.

```
import { ClientesService } from './clientes.service';
```

Como decimos, ahora este módulo ha declarado el servicio como "provider", por lo que sus componentes podrán usar este servicio.

Nota: El provider que acabamos de declarar en el módulo es un array que también podremos declarar en un componente, con lo que podríamos asignar un servicio a un componente en concreto y no a un módulo completo. Más adelante explicaremos dónde puedes declarar el provider, dependiendo de cómo quieres que tu servicio se gestione a nivel de aplicación.

Código básico de un service en Angular

Ahora podemos examinar el código generado para nuestro servicio y aprender nuevas cosas de Angular. Este sería nuestro recién creado servicio "ClientesService".

```
import { Injectable } from '@angular/core';

@Injectable()
export class ClientesService {

  constructor() {}

}
```

Como verás, el servicio no tiene nada todavía, solo su declaración, pero hay cosas interesantes

que tenemos que explicar, principalmente un nuevo decorador que no habíamos conocido hasta el momento: "@injectable".

El decorador @injectable indica a Angular que la clase que se decora, en este caso la clase ClientesService, puede necesitar dependencias que puedan ser entregadas por inyección de dependencias. De momento puedes quedarte que los servicios necesitan de este decorador, aunque realmente disponer de él no es condición indispensable.

Nota: La inyección de dependencias es un patrón de desarrollo bastante habitual en el mundo de los frameworks. Ya es familiar para los antiguos desarrolladores de Angular, pues estaba ya implementada en las versiones de AngularJS (1.x). Sin embargo, no es algo propio o específico de Angular, sino de la programación en general. Tenemos un artículo que explica de manera teórica más cosas importantes sobre este patrón de diseño de software: [Inyección de dependencias](#).

El import, arriba del todo, { Injectable } from '@angular/core', lo que hace es que nuestra clase conozca y sea capaz de usar el decorador @injectable.

Por lo demás, el servicio está vacío, pero le podemos poner ya algo de código para que nos sirva de algo. De momento vamos a exponer mediante el servicio una simple propiedad con un dato, que luego vamos a poder consumir desde algún componente. Para ello usamos las propiedades de la clase, tal como nos permite TypeScript.

```
export class ClientesService {  
  accesoFacturacion = 'https://login.example.com';  
  constructor() {}  
}
```

En nuestra clase ClientesService hemos creado una propiedad llamada "accesoFacturacion", en la que hemos asignado un valor que sería común para todos los clientes. El dato es lo de menos, lo interesante es ver que la declaración no dista de otras declaraciones en clases que hayamos visto ya.

Cómo inyectar dependencias de servicios

Ahora nos toca ver la magia de Angular y su inyección de dependencias, que vamos a usar para poder disponer del servicio en un componente.

Como en otros frameworks, en Angular la inyección de dependencias se realiza por medio del constructor. En el constructor de un componente, que hasta ahora habíamos dejado siempre vacío, podemos declarar cualquiera de los servicios que vamos a usar y el framework se encargará de proporcionarlo, sin que tengamos que realizar nosotros ningún trabajo adicional.

Esto es tan sencillo como declarar como parámetro la dependencia en el constructor del componente.

```
constructor(public clientesService: ClientesService) {}
```

De esta manera estamos indicando a TypeScript y Angular que vamos a usar un objeto "clientesService" que es de la clase "ClientesService". A partir de entonces, dentro del componente existirá ese objeto, proporcionando todos los datos y funcionalidad definida en el servicio.

Nota: Es muy importante que al declarar la inyección de dependencias en el constructor no te olvides de la declaración de visibilidad (ya sea public o private), pues si no la colocas no se generará la propiedad en el objeto construido. Esto es algo que te proporciona TypeScript, ya que en Javascript por el momento no existen los modificadores de visibilidad público o privado.

Si defines la propiedad de la clase como pública o privada, afectará a la visibilidad del servicio. En el caso que sea "public" el servicio lo podrás usar directamente desde el template. En el caso que sea "private" podrás usar el servicio únicamente dentro del código de la clase del componente.

En ambos casos "clientesService" hará que se inyecte en el constructor un objeto de la clase ClientesService.

Explicando con detalle la declaración de propiedades implícita en el constructor

Aquí tenemos que detenernos para explicar algo de la magia, o azúcar sintáctico, que te ofrece TypeScript. Porque el hecho que se declare una propiedad en un objeto solo porque se reciba un parámetro en el constructor no es algo usual en Javascript.

Cuando TypeScript detecta el modificador de visibilidad "public" o "private" en el parámetro enviado al constructor, inmediatamente declara una propiedad en la clase y le asigna el valor recibido en el constructor. Por tanto, esta declaración:

```
export class ListadoClientesComponent {  
  constructor(public clientesService: ClientesService) {}  
}
```

Sería equivalente a escribir todo el código siguiente:

```
export class ListadoClientesComponent {  
  clientesService: ClientesService;  
  constructor(clientesService: ClientesService) {  
    this.clientesService = clientesService;  
  }  
}
```

En resumen, TypeScript entiende que, si defines la visibilidad de un parámetro en el constructor, o que quieres hacer en realidad es crear una propiedad en el objeto recién construido, con el valor recibido por parámetro.

Usando el servicio en el componente

Ahora, para acabar esta introducción a los servicios en Angular, tenemos que ver cómo usaríamos este servicio en el componente. No nos vamos a detener demasiado en hacer ejemplos elaborados, que podemos abordar más adelante, solo veremos un par de muestras sobre cómo usar la propiedad declarada en el servicio.

1.- Usando el servicio dentro de la clase del componente

Dentro de la clase de nuestro componente, tendremos el servicio a partir de la propiedad usada en su declaración. En el constructor dijimos que el servicio se llamaba "clientesService", con la primera en minúscula por ser un objeto.

Pues como cualquier otra propiedad, accederemos a ella mediante la variable "this".

```
export class ListadoClientesComponent implements OnInit {  
  
  constructor(public clientesService: ClientesService) { }  
  
  ngOnInit() {  
    console.log(this.clientesService);  
  }  
  
}
```

El ejemplo no vale para mucho, solo para mostrar que, desde que esté creado el objeto podemos acceder al servicio con "this.clientesService".

También nos sirve para recordar que, el primer sitio donde podríamos usar los servicios declarados es en el método ngOnInit(). Dicho de otro modo, si necesitamos de estos servicios para inicializar propiedades en el componente, el lugar donde ponerlos en marcha sería el ngOnInit().

2.- Usando el servicio en el template de un componente

Por su parte, en el template de un componente, podrás también acceder al servicio, para mostrar sus propiedades o incluso invocar sus métodos como respuesta a un evento, por ejemplo.

Como el servicio está en una propiedad del componente, podremos acceder a él mediante ese nombre de propiedad. Pero ten en cuenta, que este acceso desde el template sólo será posible si el servicio se declaró con visibilidad "public".

```
<p>  
  URL De acceso: {{clientesService.accesoFacturacion}}  
</p>
```

Conclusión a la introducción a los servicios en Angular

Eso es todo lo que tienes que saber para comenzar a experimentar con los servicios en Angular. Es momento que pongas las manos en el código y comiences a experimentar por tu cuenta,

creando tus propios servicios y usándolos desde tus componentes.

En artículos posteriores veremos servicios un poco más completos, que hagan más cosas que el que hemos visto en la presente entrega. Puedes seguir aprendiendo a [usar clases e interfaces en los servicios de Angular](#).

Este artículo es obra de *Miguel Angel Alvarez*
Fue publicado / actualizado en 14/05/2020
Disponible online en <https://desarrolloweb.com/articulos/servicios-angular.html>

Usar clases e Interfaces en los servicios Angular

Al desarrollar servicios con Angular es una buena idea usar clases e Interfaces para definir las estructuras de datos. Veremos cómo y por qué.

Si algo caracteriza a TypeScript, el lenguaje con el que se desarrolla en Angular, es el uso de tipos. Podemos usar tipos primitivos en variables, lo que nos ayudará a recibir ayudas en el momento en el que desarrollamos el código, pero también podemos definir tipos más complejos por medio de clases e interfaces.

A lo largo del [Manual de Angular](#) hemos declarado variables indicando sus tipos en muchas ocasiones, pero hasta ahora no hemos usado casi nunca las clases para tipar y mucho menos las interfaces. En los servicios es un buen lugar para empezar a acostumbrarnos a ello, lo que nos reportará muchos beneficios a la hora de programar.

Este artículo no avanzará tanto en lo que es ofrecer nuevos conocimientos de Angular, sino más bien en el uso de costumbres que nos ayudarán en el día a día y que por tanto son muy comunes a la hora de desarrollar aplicaciones profesionales. Para entenderlo es bueno que conozcas los [fundamentos de TypeScript](#) y también específicamente de las [interfaces TypeScript](#). Obviamente, necesitarás también saber lo [que es un servicio y cómo crearlos y usarlos en Angular](#).



Ayudas en tiempo de desarrollo, no en tiempo de ejecución

Aunque puede resultar de cajón para muchas personas, queremos comenzar por señalar que las ayudas que te ofrece TypeScript serán siempre en tiempo de desarrollo. Para poder

aprovecharte de ellas lo ideal es disponer de un editor que entienda TypeScript, mostrando en el momento que desarrollas los problemas detectados en el código.

A la hora de compilar la aplicación, para su ejecución en el navegador, también te ayudará TypeScript, especificando los errores que ha encontrado en el código, derivados por un uso incorrecto de las variables y sus tipos.

Sin embargo, una vez que el navegador ejecuta el código debes entender que todo lo que interpreta es Javascript, por lo que los tipos no interferirán en nada. No será más pesado para el navegador, ni te alertará de problemas que se puedan producir, ya que éste solo ejecutará código Javascript. Sin embargo, lo cierto es que si tipaste correctamente todo lo que estuvo en tu mano y el compilador no te alertó de ningún error, difícilmente se producirá un error por un tipo mal usado en tiempo de ejecución.

Si ya tienes cierta experiencia con TypeScript habrás observado la cantidad de errores que se detectan prematuramente en el código y lo mucho que el compilador te ayuda a medida que estás escribiendo. Esto facilita ahorrar mucho tiempo y disfrutar de una experiencia de programación más agradable. Por todo ello, usar los tipos siempre que puedas es una idea estupenda y en este artículo queremos explicarte cómo ir un poco más allá, en el marco de los servicios de Angular.

Crear una clase para definir el tipo de tus objetos

La manera más común de definir tipos, para las personas acostumbradas a lenguajes de programación orientados a objetos, son las clases.

Si tienes en tu aplicación que trabajar con cualquier tipo de entidad, es una buena idea que crees una clase que especifique qué datos contiene esa estructura.

Si nuestra aplicación usa clientes, lo más normal es que definas la clase cliente, más o menos como esto:

```
class Cliente {  
  nombre: String;  
  cif: String;  
  direccion: String;  
  creado: Date;  
}
```

Luego, cuando quieras crear un cliente podrás declararlo usando el tipo de la clase Cliente.

```
let cliente: Cliente;
```

A partir de ahora, en el caso que uses un código que no respete esa declaración se alertará convenientemente.

```
class Cliente {
  nombre: String;
  cif: String;
  direccion: String;
  creado: Date;
}

let cliente: Cliente;

[ts]
Type '{ name: string; }' is not assignable to type 'Cliente'.
  Object literal may only specify known properties, and 'name' does
  not exist in type 'Cliente'.
(property) name: string

cliente = { name: 'test' }
```

Crear una interfaz para definir el tipo de tus objetos

Más o menos lo mismo podemos conseguir con una interfaz de TypeScript. Es algo que ya hemos explicado en el artículo de Interfaces TypeScript. Pero en resumen, puedes definir una interfaz de clientes de esta manera.

```
interface Cliente {
  nombre: String;
  cif: String;
  direccion: String;
  creado: Date;
}
```

Luego podemos crear una variable asignando la interface como si fuera un tipo.

```
let cliente: Cliente;
```

A partir de aquí el editor nos avisará cuando el valor asignado no cumpla la interfaz.

```
interface Cliente {
  nombre: String;
  cif: String;
  direccion: String;
  creado: Date;
}

let cliente: Cliente;

[ts]
Type '{ name: string; }' is not assignable to type 'Cliente'.
  Object literal may only specify known properties, and 'name' does
  not exist in type 'Cliente'.
(property) name: string

cliente = { name: 'test' }
```

¿Clases o interfaces?

Si ambas construcciones te sirven para más o menos lo mismo ¿cuándo usar clases y cuándo usar interfaces? la respuesta depende un poco sobre cómo vayas a generar los datos.

Es muy habitual usar simplemente interfaces, desprovistas de inicialización y funcionalidad, ya que esas partes es común que sean delegadas en los servicios. Pero en el caso de usar clases, tus nuevos objetos serán creados con la palabra "new".

```
let cliente1 = new Cliente();
cliente.nombre = 'EscuelaIT S.L.';
cliente.cif = 'B123';
cliente.direccion = 'C/ Del Desarrollo Web .com';
cliente.creado = new Date(Date.now());
```

Si los objetos que debes crear presentan tareas de inicialización pesadas, usando clases, podrás apoyarte en los constructores para resumir los pasos para su creación. En este caso podemos conseguir un código más compacto:

```
let cliente1 = new Cliente('EscuelaIT S.L.', 'B123', 'C/ Del Desarrollo Web .com');
```

Nota: obviamente, para que esto funcione tienes que haber creado el correspondiente constructor en la implementación de la clase Cliente. Observa que en caso de tener un constructor podríamos ahorrarnos pasarle el objeto de la clase Date, para asignar en la propiedad "creado", puesto que podríamos delegar en el constructor la tarea de instanciar un objeto Date con la fecha con el instante actual.

Sin embargo, los objetos que contienen datos para representar en tu aplicación no siempre se generan mediante tu propio código frontend, ya que es habitual que esos datos provengan de algún web service (API REST o similar) que te los proporcione por medio de llamadas "HTTP" (Ajax). En estos casos no harás "new", sino que usarás Angular para solicitar al servidor un dato, que será devuelto en forma de un objeto JSON habitualmente. En estos casos es especialmente idóneo definir una interfaz y declarar la el objeto que te devuelva el servidor con el tipo definido por esa interfaz. Más adelante veremos ejemplos de este caso en concreto.

Uses o no servicios web para traerte los datos, con interfaces también puedes crear objetos que correspondan con el tipo de datos definido en la interfaz. Simplemente los crearías en tus servicios por medio de literales de objeto.

```
let cliente: Cliente;
cliente = {
  nombre: 'EscuelaIT',
  cif: 'ESB123',
  direccion: 'C/ de arriba, 1',
  creado: new Date(Date.now())
};
```

Como puedes ver, la decisión sobre usar clases o interfaces puede depender de las necesidades de tu aplicación, o incluso de tus preferencias o costumbres de codificación.

Incluso, nada te impide tener ambas cosas, una interfaz y una clase implementando esa interfaz, para disponer también de la posibilidad de instanciar objetos ayudados por un constructor.

```
export interface ClienteInterface {  
  nombre: String;  
  cif: String;  
  direccion: String;  
  creado: Date;  
}  
  
export class Cliente implements ClienteInterface {  
  creado: Date;  
  constructor(public nombre: string, public cif: String, public direccion: String) {  
    this.creado = new Date(Date.now());  
  }  
}
```

Definir el modelo de datos en un archivo externo

Para ordenar el código de nuestra aplicación es también habitual que el modelo de datos se defina en un archivo TypeScript aparte. En ese archivo puedes guardar la declaración del tipo y luego importarlo en cualquier lugar donde quieras usar ese tipo de datos.

Archivo "clientes.modelo.ts"

Por ejemplo, tendríamos el archivo "clientes.modelo.ts" y ese archivo tendría, por ejemplo la declaración de la interfaz:

```
export interface Cliente {  
  nombre: String;  
  cif: String;  
  direccion: String;  
  creado: Date;  
}
```

Nota: no te olvides de colocar la palabra "export" delante del nombre de la interfaz, para que esa declaración se pueda importar desde otros archivos.

Archivo "clientes.service.ts"

Por su parte, en el servicio tendríamos que hacer el import de este tipo, definido en clientes.modelo.ts, y usarlo al declarar objetos.

```
import { Cliente } from './cliente.modelo';
```

Por supuesto, una vez definido este tipo de datos, gracias a la correspondiente interfaz

importada, lo debes de usar en tu servicio, en el mayor número de lugares donde puedas, lo que te proporcionará un código más robusto, capaz de advertirte de cualquier tipo de problemas en tiempo de desarrollo y ahorrarte muchas complicaciones.

Este sería el código de nuestro servicio, donde hacemos uso del tipo Cliente importado.

```
import { Injectable } from '@angular/core';
import { Cliente } from './cliente.modelo';

@Injectable()
export class ClientesService {

  clientes: Cliente[] = [];

  constructor() {}

  anadirCliente(cliente: Cliente) {
    this.clientes.push(cliente);
  }

  clienteNuevo(): Cliente {
    return {
      nombre: 'DesarrolloWeb.com',
      cif: 'B123',
      direccion: 'Oficinas de EscuelaIT, C/ Formación online nº 1',
      creado: new Date(Date.now())
    };
  }
}
```

En el código anterior debes fijarte en varias cosas sobre TypeScript:

- La declaración del array "clientes" indica que sus elementos son de tipo Cliente.
- En el parámetro del método "anadirCliente" hemos declarado el tipo de datos que recibimos, de tipo Cliente.
- El valor de retorno del método clienteNuevo() se ha declarado que será de tipo Cliente.

Obviamente, si durante la escritura de tu código, no solo en este servicio sino también en cualquier otro lugar donde se use, no envías objetos de los tipos esperados, TypeScript se quejará y te lo hará saber.

Fíjate en la siguiente imagen. Es el código de un componente llamado "AltaClienteComponent". Hemos cometido la imprudencia de declarar un cliente como de tipo String (flecha roja). Por ello, todos los métodos en los que trabajamos con ese cliente, apoyándonos en el servicio ClientesService, están marcados como si fueran un error (flechas amarillas).

```

import { ClientesService } from '../clientes.service';
import { Component, OnInit } from '@angular/core';

@Component({
  selector: 'dw-alta-cliente',
  templateUrl: './alta-cliente.component.html',
  styleUrls: ['./alta-cliente.component.css']
})
export class AltaClienteComponent implements OnInit {

  cliente: string;
  constructor(private clientesService: ClientesService) { }

  ngOnInit() {
    this.cliente = this.clientesService.clienteNuevo();
  }

  agregarCliente() {
    this.clientesService.anadirCliente(this.cliente);
    this.cliente = this.clientesService.clienteNuevo();
  }
}

```



Nota: La anterior imagen pertenece al editor Visual Studio Code, que ya incorpora de casa todo lo necesario para ayudarte al programar con TypeScript, mostrando los correspondientes errores encontrados en tiempo de desarrollo.

Conclusión

Con esto hemos llegado a un ejemplo de un servicio un poco más complejo del que vimos en el artículo anterior, que también se comportará de un modo más robusto, alertando de posibles problemas a la hora de escribir el código, y cuando el compilador de TypeScript analice el código para convertirlo a Javascript.

También nos ha dado pie a usar algunas de las cosas que nos aporta TypeScript, para seguir aprendiendo este superset de Javascript. Espero que a partir de ahora puedas esforzarte por sacar mayor partido a este lenguaje.

Este artículo es obra de *Miguel Angel Alvarez*
 Fue publicado / actualizado en 14/12/2017
 Disponible online en <https://desarrolloweb.com/articulos/clases-interfaces-servicios->


Práctica Angular con Módulos, Componentes y Servicios

Este es un ejercicio práctico para repasar los puntos más importantes vistos hasta el momento en el Manual de Angular. Practicaremos con una aplicación en la que trabajaremos con módulos, componentes y servicios de Angular.

```
constructor(private clientesService: ClientesService) { }

ngOnInit() {
  this.cliente = this.clientesService.nuevoCliente();
  this.grupos = this.clientesService.getGrupos();
}

nuevoCliente(): void {
  this.clientesService.agregarCliente(this.cliente);
}
```



Somos conscientes de que lo que hemos visto hasta este punto del [Manual de Angular 2](#) puede resultar complicado de asimilar si no realizamos un ejemplo completo, que nos ayude a ver de manera global el flujo de desarrollo con este framework. Por ello, vamos a pararnos aquí para practicar un poco.

Veremos cómo trabajar en el desarrollo de una aplicación Angular en la que participan todos los integrantes que hemos explicado hasta ahora, es decir: [módulos](#), [componentes](#) y [servicios](#).

En el presente ejercicio vamos a construir un sistema de alta de clientes y un listado de clientes que irá incrementando ítems, a medida que los demos de alta. Será interesante para los lectores, pero no deja de ser un sencillo demo de cómo trabajar en Angular en estos primeros pasos, pues hay mucho más en el framework que no hemos tenido tiempo de aprender todavía.

Así que vamos a poner manos a la obra. Procuraré ser muy específico, yendo paso a paso. Confío no olvidarme de mencionar ninguna parte del proceso. Ten en cuenta además que solamente resumiré algunos detalles del código, pues en artículos pasados del presente manual ha quedado ya explicado todo lo que vamos a ver.

Iremos presentando listados de cada una de las partes del código que se irán realizando y al final de este artículo encontrarás también el enlace al repositorio con el código completo.

Creamos nuestra aplicación

El primer paso es, usando el CLI, crear nuestra aplicación nueva. Lo haces entrando en la carpeta de tus proyectos y ejecutando el comando.

```
ng new clientes-app
```

Luego puedes entrar en la carpeta de la aplicación y lanzar el servidor web de desarrollo, para

ver lo que se ha construido hasta el momento.

```
cd clientes app
ng serve -o
```

Creamos nuestro módulo de clientes

La aplicación recién generada ya contiene un módulo principal, sin embargo, yo prefiero dejar ese módulo con pocas o ninguna cosa más de las que nos entregan por defecto al generar la aplicación básica. Por ello crearemos como primer paso un módulo nuevo, llamado "ClientesModule".

Encargamos a Angular CLI la creación del esqueleto de nuestro módulo con el siguiente comando:

```
ng generate module clientes
```

Definir el modelo de datos

Vamos a comenzar nuestro código por definir los tipos de datos que vamos a usar en esta aplicación. Vamos a trabajar con clientes y grupos.

Crearemos el modelo de datos dentro de la carpeta del módulo "clientes". No existe en Angular un generador de este tipo de modelos, por lo que crearé el archivo a mano con el editor. Lo voy a nombrar "cliente.model.ts".

En este archivo coloco las interfaces de TypeScript que definen los datos de mi aplicación.

```
export interface Cliente {
  id: number;
  nombre: string;
  cif: string;
  direccion: string;
  grupo: number;
}

export interface Grupo {
  id: number;
  nombre: string;
}
```

Como ves, he creado el tipo de datos Cliente y, por complicarlo un poquito más, el tipo de datos Grupo. Así, cada cliente generado pertenecerá a un grupo.

Nota: Esta parte de la creación de interfaces es perfectamente opcional. Solo la hacemos para usar esos tipos en la declaración de variables. El compilador de TypeScript nos avisará si en algún momento no respetamos estos tipos de datos, ayudando en tiempo de desarrollo y ahorrando algún que otro error derivado por despistes.

Crear un servicio para los clientes

Lo ideal es crear un servicio (service de Angular) donde concentremos las tareas de trabajo con los datos de los clientes, descargando de código a los componentes de la aplicación y centralizando en un solo archivo la lógica de la aplicación.

El servicio lo vamos a crear dentro de la carpeta del módulo clientes, por lo que especificamos la ruta completa.

```
ng generate service clientes/clientes
```

En el servicio tengo que hacer el import del modelo de datos, interfaces de Cliente y Grupo (creadas en el paso anterior).

```
import { Cliente, Grupo } from './cliente.model';
```

Nuestro servicio no tiene nada del otro mundo. Vamos a ver su código y luego explicaremos algún que otro punto destacable.

```
import { Injectable } from '@angular/core';
import { Cliente, Grupo } from './cliente.model';

@Injectable()
export class ClientesService {

  private clientes: Cliente[];
  private grupos: Grupo[];

  constructor() {
    this.grupos = [
      {
        id: 0,
        nombre: 'Sin definir'
      },
      {
        id: 1,
        nombre: 'Activos'
      },
      {
        id: 2,
        nombre: 'Inactivos'
      },
      {
        id: 3,
        nombre: 'Deudores'
      },
    ];
    this.clientes = [];
  }

  getGrupos() {
    return this.grupos;
  }

  getClientes() {
    return this.clientes;
  }
}
```



```

agregarCliente(cliente: Cliente) {
  this.clientes.push(cliente);
}

nuevoCliente(): Cliente {
  return {
    id: this.clientes.length,
    nombre: "",
    cif: "",
    direccion: "",
    grupo: 0
  };
}
}

```

1. Las dos propiedades del servicio contienen los datos que va a mantener. Sin embargo, las hemos definido como privadas, de modo que no se puedan tocar directamente y tengamos que usar los métodos del servicio creados para su acceso.
2. Los grupos los construyes con un literal en el constructor. Generalmente los traerías de algún servicio REST o algo parecido, pero de momento está bien para empezar.
3. Agregar un cliente es un simple "push" al array de clientes, de un cliente recibido por parámetro.
4. Crear un nuevo cliente es simplemente devolver un nuevo objeto, que tiene que respetar la interfaz, ya que en la función nuevoCliente() se está especificando que el valor de devolución será un objeto del tipo Cliente.
5. Fíjate que en general está todo tipado, tarea opcional pero siempre útil.

Declarar el servicio para poder usarlo en los componentes

Una tarea fundamental para poder usar los servicios es declararlos en el "module" donde se vayan a usar.

Para añadir el servicio en el module "clientes.module.ts", el primer paso es importarlo.

```
import { ClientesService } from './clientes.service';
```

Luego hay que declararlo en el array "providers".

```
providers: [
  ClientesService
]
```

Crear componente que da de alta clientes

Vamos a continuar nuestra práctica creando un primer componente. Es el que se encargará de dar de alta los clientes.

Generamos el esqueleto usando el Angular CLI.

```
ng generate component clientes/altaCliente
```

Comenzaremos editando el archivo del componente y luego iremos a trabajar con el template. Por tanto, vamos a abrir el fichero "alta-cliente.component.ts".

Agregar el servicio al componente

Muy importante. Para poder usar el servicio anterior, tengo que agregarlo al componente recién creado, realizando el correspondiente import.

```
import { ClientesService } from '../clientes.service';
```

Y posteriormente ya podré inyectar el servicio en el constructor del componente.

```
constructor(private clientesService: ClientesService) { }
```

Agregar el modelo de datos

Para poder seguir usando los tipos de datos de mi modelo, vamos a agregar el archivo donde se generaron las interfaces.

```
import { Cliente, Grupo } from '../cliente.model';
```

Código TypeScript completo del componente

El código completo de "alta-cliente.component.ts", para la definición de mi componente, quedaría más o menos así

```
import { Cliente, Grupo } from '../cliente.model';
import { ClientesService } from '../clientes.service';
import { Component, OnInit } from '@angular/core';

@Component({
  selector: 'app-alta-cliente',
  templateUrl: './alta-cliente.component.html',
  styleUrls: ['./alta-cliente.component.css']
})
export class AltaClienteComponent implements OnInit {

  cliente: Cliente;
  grupos: Grupo[];

  constructor(private clientesService: ClientesService) { }

  ngOnInit() {
    this.cliente = this.clientesService.nuevoCliente();
    this.grupos = this.clientesService.getGrupos();
  }

  nuevoCliente(): void {
    this.clientesService.agregarCliente(this.cliente);
    this.cliente = this.clientesService.nuevoCliente();
  }
}
```

```
}  
}
```

Es importante mencionar estos puntos.

1. El componente declara un par de propiedades, el cliente y el array de grupos.
2. En el constructor, que se ejecuta lo primero, conseguimos una instancia del servicio de clientes, mediante la inyección de dependencias.
3. Posteriormente se ejecuta ngOnInit(). En este punto ya se ha recibido el servicio de clientes, por lo que lo puedo usar para generar los valores que necesito en las propiedades del componente.
4. El método nuevoCliente() es el que se ejecutará cuando, desde el formulario de alta, se produzca el envío de datos. En este código usamos el servicio clientesService, para agregar el cliente y generar un cliente nuevo, para que el usuario pueda seguir dando de alta clientes sin machacar los clientes anteriormente creados.

Template del componente, con el formulario de alta de cliente

Vamos a ver ahora cuál es el HTML del componente de alta de clientes, que básicamente contiene un formulario.

Pero antes de ponernos con el HTML, vamos a hacer una importante tarea. Consiste en declarar en el módulo de clientes que se va a usar la directiva "ngModel". Para ello tenemos que hacer dos pasos:

En el archivo "clientes.module.ts" comenzamos por importar "FormsModule".

```
import { FormsModule } from '@angular/forms';
```

En el decorador, indicamos el imports del FormsModule.

```
imports: [  
  CommonModule,  
  FormsModule  
],
```

Ahora veamos el código del template, en el que reconocerás el uso de la propiedad "cliente" declarada en el constructor, así como el array de grupos.

```
<h2>Alta cliente</h2>  
<p>  
  <span>Nombre:</span>  
  <input type="text" [(ngModel)]="cliente.nombre">  
</p>  
<p>  
  <span>CIF:</span>  
  <input type="text" [(ngModel)]="cliente.cif">  
</p>  
<p>  
  <span>Dirección:</span>
```

```

<input type="text" [(ngModel)]="cliente.direccion">
</p>
<p>
<span>Grupo:</span>
<select [(ngModel)]="cliente.grupo">
  <option *ngFor="let grupo of grupos" value="{{grupo.id}}">{{grupo.nombre}}</option>
</select>
</p>
<p>
<button (click)="this.nuevoCliente()">Guardar</button>
</p>

```

Usar el componente Alta cliente

Este componente, de alta de clientes, lo quiero usar desde el componente raíz de mi aplicación. Como el componente raíz está declarado en otro módulo, necesito hacer que conozca al `AltaClienteComponent`. Esto lo consigo en dos pasos:

1.- Agregar al exports `AltaClienteComponent`

En el módulo de clientes "`clientes.module.ts`" agrego al exports el componente que quiero usar desde otros módulos.

```

exports: [
  AltaClienteComponent
]

```

2.- Importar en el módulo raíz

Ahora, en el módulo raíz, "`app.module.ts`", debes declarar que vas a usar componentes que vienen de `clientes.module.ts`. Para ello tienes que hacer el correspondiente import:

```

import { ClientesModule } from './clientes/clientes.module';

```

Y luego declaras el módulo en el array de imports:

```

imports: [
  BrowserModule,
  ClientesModule
],

```

Hechos los dos pasos anteriores, ya puedes usar el componente en el template. Para ello simplemente tenemos que escribir su tag, en el archivo "`app.component.html`".

```

<app-alta-cliente></app-alta-cliente>

```

Llegado a este punto, si todo ha ido bien, deberías ver ya el componente de alta de clientes funcionando en tu página.

Nota: Si se produce cualquier error, entonces te tocará revisar los pasos anteriores o hacer una búsqueda con el texto del error que te devuelva la consola del navegador o el terminal, para ver dónde te has equivocado.

Crear el componente listado-cliente

Para acabar nuestra práctica vamos a crear un segundo componente de aplicación. Será el componente que nos muestre un listado de los clientes que se van generando.

Como siempre, comenzamos con un comando del CLI.

```
ng generate component clientes/listadoClientes
```

Ahora el flujo de trabajo es similar al realizado para el componente anterior. Vamos detallando por pasos.

Creas los import del servicio y de los tipos de datos del modelo.

```
import { Cliente, Grupo } from '../cliente.model';  
import { ClientesService } from '../clientes.service';
```

Injectas el servicio en el constructor.

```
constructor(private clientesService: ClientesService) { }
```

En este componente tendremos como propiedad el array de clientes que el servicio vaya creando. Así pues, declaras dicho array de clientes:

```
clientes: Cliente[];
```

Cuando se inicialice el componente tienes que solicitar los clientes al servicio. Esto lo hacemos en el método `ngOnInit()`.

```
ngOnInit() {  
  this.clientes = this.clientesService.getClientes();  
}
```

Código completo del componente `ListadoClientesComponent`

Puedes ver a continuación el código TypeScript completo de cómo nos quedaría este segundo componente.

```
import { Cliente, Grupo } from '../cliente.model';
import { ClientesService } from '../clientes.service';
import { Component, OnInit } from '@angular/core';

@Component({
  selector: 'app-listado-clientes',
  templateUrl: './listado-clientes.component.html',
  styleUrls: ['./listado-clientes.component.css']
})
export class ListadoClientesComponent implements OnInit {

  clientes: Cliente[];
  constructor(private clientesService: ClientesService) { }

  ngOnInit() {
    this.clientes = this.clientesService.getClientes();
  }

}
```

Código de la vista

Ahora podemos ver cómo sería la vista, código HTML, del listado de componentes.

```
<h2>
  Listado clientes
</h2>
<div *ngIf="! clientes.length">No hay clientes por el momento</div>
<div>
  <article *ngFor="let cliente of clientes">
    <span>{{cliente.nombre}}</span>
    <span>{{cliente.cif}}</span>
    <span>{{cliente.direccion}}</span>
    <span>{{cliente.grupo}}</span>
  </article>
</div>
```

No lo hemos comentado anteriormente, pero puedes darle un poco de estilo a los componentes editando el archivo de CSS. Por ejemplo, este sería un poco de CSS que podrías colocar en el fichero "listado-clientes.component.css".

```
article {
  display: flex;
  border-bottom: 1px solid #ddd;
  padding: 10px;
  font-size: 0.9em;
}
span {
  display: inline-block;
  width: 22%;
  margin-right: 2%;
}
```

Usar el componente del listado

Para usar este componente de listado de clientes, ya que lo queremos invocar desde el módulo raíz, tienes que ampliar el exports del module "clientes.module.ts".

```
exports: [  
  AltaClienteComponent,  
  ListadoClientesComponent  
]
```

Como para el anterior componente, de alta de clientes, ya habíamos importado el módulo de clientes, no necesitas hacer nada más. Ahora ya puedes usar el usar el componente directamente en el template del componente raíz "app.component.html".

```
<app-listado-clientes></app-listado-clientes>
```

Es hora de ver de nuevo la aplicación construida y disfrutar del buen trabajo realizado. El aspecto de la aplicación que hemos realizado debería ser más o menos el siguiente:

Alta cliente

Nombre:

CIF:

Dirección:

Grupo: 

Listado clientes

Cliente 1	B 123	C/ la la la	1
Cliente 2	A 334	Av. lo lo lo	2

Conclusión

Llegado a este punto, hemos terminado la práctica. Tenemos un sistema de alta y visualización de clientes dinámico, generado en una aplicación Angular con diferentes piezas del framework.

El código completo lo puedes ver en este repositorio en GitHub, en [este enlace que te lleva a un commit concreto](#).

Si has entendido el proceso y por tanto has podido seguir los pasos, estás en el buen camino. Ya conoces las principales piezas para el desarrollo en Angular. Aún queda mucho por aprender, pero los siguientes pasos serán más sencillos.

Si te ha faltado información para entender lo que hemos hecho en este artículo, te sugiero que

leas con calma los correspondientes artículos del [Manual de Angular](#), en los que detallamos cada una de las piezas usadas en esta aplicación de demo.

Es normal también que te salgan errores sobre la marcha. Para resolverlos el compilador de TypeScript y el propio Angular ayudan bastante. Puedes preguntar o "googlear" para encontrar respuestas a tus problemas.

Este artículo es obra de *Miguel Angel Alvarez*
Fue publicado / actualizado en 14/05/2020
Disponible online en <https://desarrolloweb.com/articulos/practica-angular-modulos-componentes-servicios.html>

Observables en Angular

En los próximos artículos introduciremos el concepto de observable, una herramienta para la programación reactiva dentro de aplicaciones de Angular, capaz de aumentar sensiblemente el rendimiento de las aplicaciones.

Introducción teórica a los observables en Angular

Aclaremos conceptos sobre los observables, por qué son importantes en Angular., qué es la programación reactiva y qué es RxJS.

En este artículo vamos a comenzar una nueva etapa en nuestro conocimiento de Angular, dedicando tiempo para una primera aproximación a los "observables", que son una de las principales novedades del framework a partir de Angular 2. Los observables representan también una de las mejores formas de optimizar una aplicación, aumentando su rendimiento.

En Angular se usan mucho los observables, dada su utilidad y versatilidad, aunque hemos de admitir que no es una tarea sencilla de aprender inicialmente. Intentaremos acercármelos de una manera sencilla, para que los puedas ir digiriendo poco a poco y suavizar su curva de aprendizaje. De momento, en este artículo del [Manual de Angular](#), nuestro objetivo es ofrecer una introducción general, para que comiences a conocerlos, así como el concepto de programación reactiva.



El "por qué" de los observables

Hemos dicho que los observables son una de las principales herramientas para programar aplicaciones de mayor rendimiento. Obviamente ese es el motivo por el cuál se usan en Angular. Pero, ¿dónde reside esa mejora de rendimiento?

Uno de los motivos por los que Angular (y en especial su predecesor AngularJS) se convirtió en un framework tan usado es su capacidad de proporcionar una actualización automática de las fuentes de información. Es decir, en Angular somos capaces de usar un almacén de datos y, cuando se modifica ese almacén, recibir automáticamente sus cambios, sin que tengamos que

programar a mano ese tránsito de la información.

Incluso, aunque un componente sea el encargado de actualizar ese almacén de datos, hemos visto que, usando servicios, podemos conseguir que otros componentes reciban automáticamente las actualizaciones. Si no lo recuerdas, consulta el artículo de [práctica con Angular con componentes, módulos y servicios](#).

Sin embargo, aunque Angular nos ahorra escribir mucho código, éste tiene un coste en términos de rendimiento. Quizás una aplicación pequeña no se verá tan afectada por el trabajo que Angular hace por debajo, para proporcionarnos automáticamente los cambios, pero sí se dejará notar en aplicaciones medianas. Ya las más grandes aplicaciones acusarán sensiblemente una mayor falta de rendimiento.

Nota: Para ser más concreto, Angular por debajo hace una serie de operaciones de manera repetitiva, en las que consulta los cambios en la fuente de datos, para saber cuándo se actualizan y entonces realizar las acciones oportunas para refrescar los datos en los lugares donde se están usando. Esa no era la mejor estrategia posible y por este motivo, otras librerías como [ReactJS](#), que supieron implementar un patrón de comportamiento más acertado, capaz de ofrecer mayor rendimiento, comenzaron a ganar su espacio ante la hegemonía de Angular.

La solución aplicada en Angular 2 (y que mantienen las siguientes versiones, 4, 5...) fué usar un patrón llamado "Observable", que básicamente nos ahorra tener que hacer consultas repetitivas de acceso a la fuente de información, aumentando el rendimiento de las aplicaciones.

Programación reactiva

Hacemos aquí una pausa para introducir otro concepto relacionado con los observables, como es la "programación reactiva". Aunque para hablar de programación reactiva existen libros enteros, vamos a explicar muy por encima sobre lo que se trata.

Programación tradicional

Primero establezcamos una base sobre un conocimiento de la programación tradicional que nos parece obvio, pero que es la base sobre la programación reactiva, que tiene que ver con el flujo de ejecución de las instrucciones.

En programación tradicional las instrucciones se ejecutan una detrás de otra. Por tanto, si realizamos un cálculo con dos variables y obtenemos un resultado, aunque las variables usadas para hacer el cálculo cambien en el futuro, el cálculo ya se realizó y por tanto el resultado no cambiará.

```
let a = 1;
let b = 3;
let resultado = a + b; // resultado vale 4
// Más tarde en las instrucciones...
a = 7; // Asignamos otro valor a la variable a
```

```
// Aunque se cambie el valor de "a", resultado sigue valiendo 4,
```

El anterior código ilustra el modo de trabajo de la programación tradicional y la principal diferencia con respecto a la programación reactiva. Aunque pueda parecer magia, en programación reactiva la variable resultado habría actualizado su valor al alterarse las variables con las que se realizó el cálculo.

Programación reactiva y los flujos de datos

Para facilitar el cambio de comportamiento entre la programación tradicional y la programación reactiva, en ésta última se usan intensivamente los flujos de datos. La programación reactiva es la programación con flujos de datos asíncronos.

En programación reactiva se pueden crear flujos (streams) a partir de cualquier cosa, como podría ser los valores que una variable tome a lo largo del tiempo. Todo puede ser un flujo de datos, como los clics sobre un botón, cambios en una estructura de datos, una consulta para traer un JSON del servidor, un feed RSS, el listado de tuits de las personas a las que sigues, etc.

En la programación reactiva se tienen muy en cuenta esos flujos de datos, creando sistemas que son capaces de consumirlos de distintos modos, fijándose en lo que realmente les importa de estos streams y desechando lo que no. Para ello se dispone de diversas herramientas que permiten filtrar los streams, combinarlos, crear unos streams a partir de otros, etc.

Como objetivo final, reactive programming se ocupa de lanzar diversos tipos de eventos sobre los flujos:

- La aparición de algo interesante dentro de ese flujo
- La aparición de un error en el stream
- La finalización del stream

Como programadores, mediante código, podemos especificar qué es lo que debe ocurrir cuando cualquiera de esos eventos se produzca.

Si quieres leer más sobre programación reactiva, una introducción mucho más detallada la encuentras en el artículo [The introduction to Reactive Programming you've been missing](#).

Observables y programación reactiva

El patrón observable no es más que un modo de implementación de la programación reactiva, que básicamente pone en funcionamiento diversos actores para producir los efectos deseados, que es reaccionar ante el flujo de los distintos eventos producidos. Mejor dicho, producir dichos eventos y consumirlos de diversos modos.

Los componentes principales de este patrón son:

- **Observable:** Es aquello que queremos observar, que será implementado mediante una colección de eventos o valores futuros. Un observable puede ser creado a partir de eventos de usuario derivados del uso de un formulario, una llamada HTTP, un almacén

de datos, etc. Mediante el observable nos podemos suscribir a eventos que nos permiten hacer cosas cuando cambia lo que se esté observando.

- **Observer:** Es el actor que se dedica a observar. Básicamente se implementa mediante una colección de funciones callback que nos permiten escuchar los eventos o valores emitidos por un observable. Las callbacks permitirán especificar código a ejecutar frente a un dato en el flujo, un error o el final del flujo.
- **Subject:** es el emisor de eventos, que es capaz de crear el flujo de eventos cuando el observable sufre cambios. Esos eventos serán los que se consuman en los observers.

Estas son las bases del patrón. Sabemos que hemos puesto varios conceptos que sólo quedarán más claros cuando los veamos en código. Será dentro de poco. Aunque vistas muy por encima, son conceptos con los que merece la pena comenzar a familiarizarse.

Existen diversas librerías para implementar programación reactiva que hacen uso del patrón observable. Una de ellas es RxJS, que es la que se usa en Angular.

Qué es RxJS

Reactive Extensions (Rx) es una librería hecha por Microsoft para implementar la programación reactiva, creando aplicaciones que son capaces de usar el patrón observable para gestionar operaciones asíncronas. Por su parte RxJS es la implementación en Javascript de ReactiveExtensions, una más de las adaptaciones existentes en muchos otros lenguajes de programación.

RxJS nos ofrece una base de código Javascript muy interesante para programación reactiva, no solo para producir y consumir streams, sino también para manipularlos. Como es Javascript la puedes usar en cualquier proyecto en este lenguaje, tanto del lado del cliente como del lado del servidor.

La librería RxJS de por sí es materia de estudio para un curso o un manual, pero tenemos que introducirla aquí porque la usa Angular para implementar sus observables. Es decir, en vez de reinventar la rueda, Angular se apoya en RxJS para implementar la programación reactiva, capaz de mejorar sensiblemente el desempeño de las aplicaciones que realicemos con este framework.

Como habrás entendido, podemos usar RxJS en diversos contextos y uno de ellos son las aplicaciones Angular. En los próximos artículos comenzaremos a ver código de Angular para la creación de observables y de algún modo estaremos aprendiendo la propia librería RxJS.

Conclusión

Con lo que hemos tratado en este artículo tienes una base de conocimiento esencial, necesaria para dar el paso de enfrentarte a los observables en Angular.

No hemos visto nada de código pero no te preocupes, porque en el próximo artículo vamos a realizar una práctica de uso de observables en Angular con la que podrás practicar con este modelo de trabajo para la comunicación de cambios. Lo importante por ahora es aclarar conceptos y establecer las bases de conocimiento necesarias para que, a la hora de ver el

código, tengas una mayor facilidad de entender cómo funciona esto de los observables y la programación reactiva.

Este artículo es obra de *Miguel Angel Alvarez*
Fue publicado / actualizado en 16/01/2018
Disponible online en <https://desarrolloweb.com/articulos/introduccion-teorica-observables-angular.html>

Práctica de observables en Angular

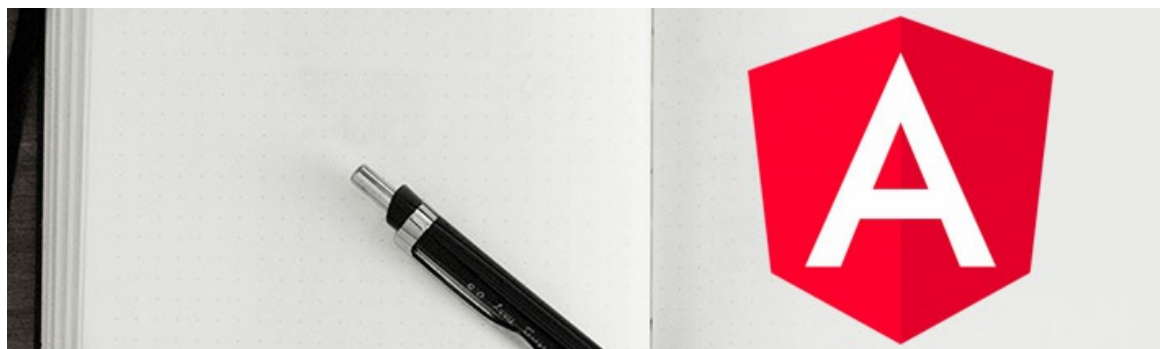
Primeros pasos trabajando con observables en una aplicación desarrollada con Angular.

En el artículo anterior del [Manual de Angular](#) ofrecimos una [introducción teórica a todos los conceptos relacionados con los observables](#). Ahora nos vamos a poner manos a la obra para ver cómo se traduce todo esto en código, mediante un sencillo ejemplo práctico.

En este artículo trabajaremos con los observables, usando la librería RxJS, que es aquella en la que se apoya Angular para introducir este patrón de desarrollo, capaz de aumentar sensiblemente el desempeño de las aplicaciones.

Para esta aplicación práctica vamos a partir de un ejemplo que ya veníamos trabajando anteriormente en este manual. Para que nadie se pierda, dejamos la referencia al ejemplo sobre el que vamos a aplicar los observables: [práctica de módulos, componentes y servicios en el sistema de alta y listado de clientes](#).

En ese pasado artículo pudimos desarrollar dos componentes, que compartían un mismo almacén de datos (un array de objetos cliente). En el componente de alta se generaban nuevos objetos y se introducían en el array y en el componente de listado se mostraban. Nuestro servicio era el encargado de mantener ese almacén de datos y servir de puente de comunicación entre ambos componentes. El ejemplo funcionaba perfectamente y en aplicaciones pequeñas no necesitamos nada más. Pero si tenemos una aplicación grande, o una mediana donde necesitemos mantener un mejor desempeño, aplicar los observables será una opción mucho mejor.



Qué necesitamos para implementar observables

Si leíste con atención el artículo anterior, dedicado a [introducir los observables](#), creo que más o

menos puedes tener una idea de lo que vamos a necesitar en este paso de implementación. De todos modos, nunca está de más un pequeño repaso para aclarar las ideas. En resumen, estos son los actores que vamos a usar:

- Un subject: que es el encargado de generar y emitir los eventos de actualización del almacén de datos.
- Un observable: que es un objeto que permite observar los eventos emitidos por el subject.
- Una suscripción: generada a partir del observable.

Nota: Otras implementaciones pueden poner en juego otros tipos de objetos, disponibles en el patrón observer y en la librería RxJS.

El servicio de clientes será el encargado de usar el "subject" para generar los eventos cuando se agreguen nuevos clientes. Este servicio también será el encargado de entregar el observable a los componentes que necesiten escuchar los eventos emitidos por el subject.

Aquellos componentes escuchando eventos son los observadores (observer). Estos observadores declaran y usan el observable y, mediante una suscripción, estarán enterados de cualquier cambio en aquello que se está observando.

Esperamos que estos conceptos estén medianamente claros, puesto que será esencial para poder entender el resto del ejercicio. A continuación podemos comenzar a revisar con detalle el código necesario para realizar todas estas operativas.

Producir el flujo de eventos a observar

El servicio que implementa toda la parte de acceso a los datos, con la lógica de negocio de la aplicación, es el encargado de generar los eventos que los observadores podrán consumir.

El servicio tiene por tanto que conocer la implementación de dos clases:

- El subject, para generar el stream de eventos
- El observable, que entregarán bajo demanda a los observadores

Para ello, en el servicio de clientes "clientes.service.ts" vamos a crear los correspondientes imports:

```
import { Observable } from 'rxjs/Observable';  
import { Subject } from 'rxjs/Subject';
```

Como puedes comprobar, estos import no te los ofrece la librería Angular, sino la librería RxJS, que se instaló como dependencia al crear la aplicación Angular.

Declaración del subject como propiedad

El subject será un elemento del servicio y por tanto lo tenemos que declarar como cualquier otra propiedad. Su declaración se realiza mediante el siguiente código.

```
private clientes$ = new Subject<Cliente[]>();
```

- Realizamos la declaración privada, para que nadie pueda acceder al subject, excepto el propio servicio.
- El nombre de la propiedad privada es clientes\$. Es normal que los subjects en el trabajo con observables tengan el carácter "[[--body--]]"; al final, para dejar claro lo que son.
- Usamos un "generic" de TypeScript para indicar el tipo de aquello que vamos a observar. Lo que vamos a observar es el array de clientes, por eso en la instanciación del subject se define el tipo genérico como Cliente[].
- No hace falta indicar el tipo de la propiedad series\$: Subject <cliente[]> porque se infiere perfectamente, debido a que la instanciación del subject se hace en la misma línea de la declaración de la propiedad.</cliente[]>

Nota: tienes un artículo muy completo para saber más sobre los [Generics de TypeScript](#).

Emitir eventos usando el subject

Ahora, también en el servicio, cada vez que se agregue un nuevo elemento al array de clientes, tenemos que emitir un evento, usando el subject. Esto lo conseguimos, por supuesto, usando el objeto subject de RxJS.

Así quedaría el método que agrega un cliente, que ya tenía nuestro servicio anteriormente. Solo que antes únicamente se realizaba el push() y ahora seguimos realizando la inserción en el array y posteriormente la generación del evento con el subject.

```
agregarCliente(cliente: Cliente) {  
  this.clientes.push(cliente);  
  this.clientes$.next(this.clientes);  
}
```

Como puedes ver, el método para crear el evento es "next" y además le tenemos que pasar el estado del array en este momento, para que luego los observadores puedan saber cómo estaba el array al producirse este evento.

Generar el observable

La última acción que vamos a necesitar para completar el trabajo dentro del servicio es la generación del observer, que se entregará a todos aquellos componentes que quieran observar cambios en el almacén de datos.

Es interesante este paso, puesto que el observable es un consumidor de los eventos del subject y es de sólo lectura. Es decir, puede estar atento a eventos, pero es incapaz de hacer nada más. Por tanto, lo único que el servicio entregará a externamente, a cualquier componente que lo necesite, es este observable. Mediante el observable, los componentes sabrán cuando el almacén de datos se ha modificado, pero ningún componente podrá generar nuevos eventos de cambio del almacén de datos, que es una responsabilidad del servicio y tarea principal del subject privado.

El observer se creará mediante un método del subject llamado `asObservable()`. Usaremos un método "getter" para devolver el observable, que tendrá el siguiente aspecto.

```
getClientes$(): Observable<Cliente[]> {  
  return this.clientes$.asObservable();  
}
```

Como puedes ver, el método devuelve un objeto de la clase `Observable`, el cual también se tiene que definir especificando, mediante un genérico de TypeScript, aquello que ese observable es capaz de vigilar, en este caso un array de clientes (`Clientes[]`).

Consumir un observable

Ahora, en cualquier componente que necesitemos estar atentos a los cambios del almacén de datos, podemos usar el observable que nos ofrece el servicio. Para ello necesitamos solamente un par de pasos.

El primero es importar la clase `Observable`.

```
import { Observable } from 'rxjs/Observable';
```

Seguidamente, vamos a crear una propiedad en el componente que permita almacenar el observable. Podemos observar que se declara el tipo, pero no se inicializa todavía.

```
clientes$: Observable<Cliente[]>;
```

Nota: De nuevo, para orientar a quien sea que lea este código, le colocamos un `$` al final del nombre de la variable, indicando que es un observable.

Crear la suscripción a los eventos

El segundo es crear la suscripción a los eventos que nos entrega el observable, y que fueron generados en el servicio usando el subject.

El lugar adecuado para generar esa suscripción es el método `ngOnInit()`, que se ejecuta cuando

el componente ya se ha inicializado y por tanto tiene todas sus propiedades ya disponibles. Tendrá esta forma:

```
ngOnInit() {  
  this.clientes$ = this.clientesService.getClientes$();  
  this.clientes$.subscribe(clientes => this.clientes = clientes);  
}
```

Nota: Así haremos en cualquier otro componente que necesite usar el observable. Obviamente, en el constructor del componente ListadoClientesComponent se debe haber inyectado el servicio que se necesita para pedir el observable.

En el código anterior hacemos dos pasos:

- Se accede al observable, mediante el método `getClientes$()` del servicio `clientesService`. Ese observable es el que puede escuchar los eventos que necesitamos consumir.
- Se crea una suscripción mediante el método `suscribe()` del observable. Este método de suscripción debe recibir la función manejadora de eventos que contiene el código a ejecutar cuando se dispara el evento. La función manejadora de eventos recibe el array que se está observando como parámetro.

Nota: Puedes ver que estamos usando una "arrow function" para la definición del manejador de eventos de la suscripción. Podríamos haber usado perfectamente una función anónima como quizás estás acostumbrado, pero aquí es preferible utilizar una "función flecha" por dos motivos. 1) La sintaxis es más concisa y reducida. Pero es más importante aún 2) la función flecha no genera contexto propio, por lo que podemos seguir accediendo a "this". Si lo deseas, en el [manual de ES6](#) puedes obtener mucha más información de las [arrow functions](#).

Básicamente, lo que hace el manejador de eventos de la suscripción es que, cada vez que cambia el array de clientes, actualiza la propiedad interna del componente al nuevo array actualizado, recibido por parámetro.

De este modo, el listado de clientes puede recibir el nuevo array de clientes, cada vez que se actualiza, de una manera optimizada, puesto que no tiene que preguntar todo el tiempo si ese array se ha actualizado, solo sentarse a esperar y ser notificado cada vez que hay cambios.

Después de realizar esas modificaciones, tu aplicación debe seguir funcionando igual que hasta ahora, mostrando clientes en el listado cada vez que el componente de alta de clientes nos genera uno nuevo.

Para resolver posibles dudas en cuanto al código completo del proyecto, tal como lo hemos dejado en este paso, te dejo este enlace a un commit en un repositorio del ejemplo en GitHub. https://github.com/midesweb/ej_facturacion_angular/tree/c240fode34ab21341coc85db2b85fabb

En la siguiente entrega, vamos a explicar otra cosa fundamental en este proceso, que debes aprender para completar el flujo: dar de baja las suscripciones a los observables.

Este artículo es obra de *Miguel Angel Alvarez*
Fue publicado / actualizado en 30/01/2018
Disponible online en <https://desarrolloweb.com/articulos/practica-observables-angular.html>

Eliminar la suscripción a un observable de Angular

Cómo eliminar una suscripción a un observable, desde el observador, necesario cuando el elemento se destruye para no dejar en memoria la suscripción.

En este artículo vamos a estudiar una práctica fundamental con los observables de Angular (incorporados mediante la librería RxJS), que consiste en dar de baja las suscripciones cuando ya no tiene sentido que permanezcan en memoria.

Enseguida explicaré el motivo de la importancia de este paso, que complementa las explicaciones de artículos anteriores del [Manual de Angular](#). De hecho, si leíste el pasado capítulo de este manual "[Práctica con Observables](#)" nos dejamos pendiente explicar cómo completar el flujo mediante el paso de dar de baja la suscripción a los eventos del observable.

Este artículo de paso servirá para seguir aprendiendo cosas de Angular en general, como el uso del método `ngOnDestroy()`, que nos permite ejecutar código cuando un componente deja de existir. Así que vamos con ello!



Nota: En este artículo continuaremos el código del componente `ListadoClientesComponent` realizado en el artículo anterior. Es el archivo "`listado-clientes.component.ts`". Para mayores referencias, por favor, lee el [Práctica de observables en Angular](#).

Por qué es importante eliminar las suscripciones a los observables

Al trabajar con observables creamos suscripciones a un flujo de datos o eventos. Esas suscripciones permanecen en memoria el tiempo que sean necesario, que habitualmente debe

ser el tiempo en el que un componente exista. Si el componente que está escuchando eventos o datos mediante una suscripción deja de existir, no tiene sentido que la suscripción permanezca ejecutándose en el sistema. Angular en muchas ocasiones se encarga de eliminar las suscripciones a los observables, pero en otras ocasiones podría no hacerlo y si somos suficientemente meticulosos, deberíamos realizarlo por nuestra cuenta con un poco de código.

En este punto puede que te preguntes ¿Cómo es posible que el componente deje de existir? La respuesta puede ser bien obvia, pero sólo se entenderá con el sistema de routing en mente.

En el sistema de routing puede ocurrir que estemos viendo una pantalla u otra de la aplicación, según la ruta consumida. En cada ruta generalmente hay un componente que se encarga de mostrar aquella pantalla o vista que se tiene que mostrar al usuario. Cuando el usuario pasa a otra ruta de la aplicación, ocurre que el componente se destruye, y se genera un nuevo componente en su lugar, con la vista de la nueva pantalla.

Así pues, en la destrucción de un componente, al pasar de una ruta a otra, es esencial que las suscripciones existentes se eliminen de la memoria.

Método ngOnDestroy()

El método ngOnDestroy() pertenece al ciclo de vida de los componentes de Angular. Es el lugar adecuado para realizar las acciones necesarias a ejecutar con la destrucción de un componente.

Es muy parecido al conocido ngOnInit(), pero se ejecuta cuando un componente deja de existir.

Para usar este método tenemos que implementar la interfaz OnDestroy. Y por supuesto, dentro de la implementación del componente, se debe definir el método ngOnDestroy(), para cumplir el contrato definido por esta interfaz.

Primero tenemos que asegurarnos de importar la interfaz OnDestroy, desde "@angular/core".

```
import { Component, OnInit, OnDestroy } from '@angular/core';
```

Una vez importada, ya podemos implementarla, de la siguiente manera.

```
export class MiComponent implements OnInit, OnDestroy {  
  ngOnInit() {  
    // acciones de inicialización  
  }  
  
  ngOnDestroy() {  
    // acciones de destrucción  
  }  
}
```

Cómo eliminar una suscripción a un observable

Esta parte incluye varios pasos a realizar, que pasamos a resumir.

1.- Importar la declaración de suscripción de un observable

Vamos a tener que almacenar la suscripción en algún lugar y para poder tiparla convenientemente vamos a importar la declaración de "subscription definida por RxJS".

```
import { Subscription } from 'rxjs/Subscription';
```

2.- Declarar la suscripción como propiedad del componente

Entre las propiedades del componente vas a tener que declarar la suscripción, para disponer de ella cuando sea necesaria. Es de tipo "Subscription".

```
clientesSubscription: Subscription;
```

3.- Guardarnos la suscripción en el paso de su creación

Usamos la propiedad creada en el paso anterior para almacenar el objeto de la suscripción. Simplemente asignamos el valor que nos devuelve el método suscribe() sobre el observable. Recuerda que la suscripción en nuestro ejemplo la habíamos creado en ngOnInit(), después de haber solicitado el observable al servicio.

```
ngOnInit() {  
  this.clientes$ = this.clientesService.getClientes$();  
  this.clientesSubscription = this.clientes$.subscribe(clientes => this.clientes = clientes);  
}
```

4.- Eliminamos la suscripción al destruirse el elemento

Ya por fin, una vez el componente se destruya, realizamos el paso más relevante, que es eliminar la suscripción. Lo hacemos en el método ngOnDestroy y usamos la propiedad donde habíamos guardado la suscripción. Esa suscripción tiene un método llamado unsubscribe() que hace la tarea deseada.

```
ngOnDestroy() {  
  this.clientesSubscription.unsubscribe();  
}
```

Con esto ya hemos solucionado la necesidad de eliminar la suscripción al observable. Ahora estamos seguros de que no se quedará colgada, ocupando espacio en la memoria, al destruirse el componente que la utilizaba.

No obstante, hay que decir que este componente no usaba el sistema de routing y no había modo de ser destruido durante la vida de la aplicación, por lo que en realidad este paso no era necesario para este ejemplo en concreto. Aunque tampoco es mala idea ser cuidadosos y colocar un código que produzca un componente más correcto y capaz de funcionar en todas

partes sin problemas. De hecho, si ahora implementamos un sistema de routing y usamos este componente, se comportará perfectamente en su función, no dejando colgada en memoria la suscripción al observable.

El código de esta práctica, tal como la hemos dejado después de este artículo, lo puedes ver en GitHub:

https://github.com/midesweb/ej_facturacion_angular/tree/48427d9a86c9dc83dbafo6bc9b72e35

Este artículo es obra de *Miguel Angel Alvarez*

Fue publicado / actualizado en 07/02/2018

Disponible online en <https://desarrolloweb.com/articulos/eliminar-suscripcion-observable-angular.html>

Pipes en Angular

Vamos a abordar ahora los pipes en Angular, que nos permiten realizar transformaciones de datos que queremos usar en las vistas de los componentes.

Crear y usar pipes en Angular

En este artículo vamos a mostrar cómo usar pipes en Angular, para transformar los datos y cómo crear nuestros propios pipes personalizados.



Los pipes son una de las utilidades que nos ofrece Angular para realizar transformaciones de los datos, a la hora de mostrarlos en los templates de los componentes. Son muy útiles en inmensas situaciones y usarlos es muy sencillo.

Crear pipes requiere unos conocimientos extra pero con los detalles que te vamos a explicar no vas a tener ningún problema.

Cómo usar pipes en Angular

Angular ya tiene incorporados algunos pipes de uso habitual que podemos implementar cómodamente para comenzar a experimentar con esta utilidad.

Un pipe muy útil para debugear lo que te llegan en los objetos que recibes de los servicios web es "json". Seguramente lo hayas usado ya. Dentro de un template lo puedes implementar con el carácter de tubería, indicando después de la tubería el nombre del pipe que deseas usar.

```
<pre>{{users | json}}</pre>
```

Suponiendo que "users" es un array de objetos usuario, este pipe te mostrará la cadena de su representación JSON.

Usar el pipe date

Vamos a ver un ejemplo un poco más completo de uso de pipes para mostrar fechas de manera más adecuada para nuestros usuarios. El pipe en cuestión se llama "date" y está integrado dentro de Angular.

Vamos a tener un componente que declara una fecha como propiedad, con la fecha del día, en el atributo "hoy".

```
@Component({
  selector: 'app-usar-pipes',
  templateUrl: './usar-pipes.component.html',
  styleUrls: ['./usar-pipes.component.css']
})
export class UsarPipesComponent implements OnInit {

  hoy = new Date();

  constructor() {}

  ngOnInit(): void {
  }

}
```

Ahora, en el template del componente, vamos a mostrar la fecha con el pipe date, de esta manera:

```
<p>Hoy es {{ hoy | date }}</p>
```

Enviar datos a los pipes

Los pipes pueden recibir datos adicionales para realizar su trabajo de manera determinada. Para ello indicamos el nombre del pipe, seguido de el caracter dos puntos ":" y luego el dato que le queremos pasar al pipe.

El pipe "date" permite indicarle como dato extra el formato de la fecha que queremos mostrar, por ejemplo para indicarle que la fecha la muestre tal como estamos acostumbrados en español.

```
<p>Hoy es {{ hoy | date:"dd/MM/yy" }}</p>
```

Cómo crear nuestros propios pipes

Con lo que hemos visto ya conoces lo básico para usar pipes en Angular. Sin embargo la gracia es que puedas crear pipes en Angular por ti mismo, para poder hacer las transformaciones que sean necesarias en tus aplicaciones.

En este ejemplo vamos a hacer un pipe que permita filtrar usuarios de un array de usuarios que podemos recibir desde un API cualquiera.

Generar el scaffolding del pipe

Para crear un pipe comenzamos generando el esqueleto de la clase, con el comando siguiente:

```
ng generate pipe nombre
```

A continuación podemos ver cómo, además de crearse dos archivos "nombre.pipe.ts" y "nombre.pipe.spec.ts", se ha editado el módulo de la aplicación, generando el código necesario para usar el pipe en ella.

En app.module.ts encontrarás el import del pipe:

```
import { NombrePipe } from './nombre.pipe';
```

Además en el array declarations estará incluido el pipe recién creado.

```
declarations: [  
  AppComponent,  
  NombrePipe,  
  // ...  
],
```

Crear el código para realizar la transformación del pipe

Ahora vamos a ver el código de nuestro pipe, que nos sirve para filtrar usuarios de un array, encontrando coincidencias de una cadena de búsqueda en su nombre.

Todo pipe tiene un método "transform", que recibe siempre el valor de entrada del pipe. Adicionalmente recibirá datos extra que el pipe necesite para operar. En nuestro caso como valor de entrada recibiremos un array y como dato extra recibiremos la cadena que se desea buscar en el atributo nombre de los objetos de ese array.

Veamos el código del pipe:

```
import { Pipe, PipeTransform } from '@angular/core';  
  
@Pipe({  
  name: 'nombre'  
})  
export class NombrePipe implements PipeTransform {  
  
  transform(value: any[], query: string): unknown {  
    if(query === '' || query === undefined) {  
      return value;  
    }  
  }  
}
```

```

    }
    return value.filter(user => user.name.toLowerCase().indexOf(query) !== -1)
  }
}

```

Como puedes ver, lo primero que hacemos es comprobar si la consulta recibida en el parámetro query está vacía o es igual a undefined. En ese caso devolvemos el objeto de usuarios, tal cual.

Si tenemos alguna consulta en el parámetro query, entonces realizamos el filtrado de usuarios mediante la búsqueda de esos caracteres en el atributo "name". Devolvemos finalmente el array filtrado.

Usar el pipe que hemos creado

Ahora vamos a usar ese pipe. Para ello tenemos un componente que recibe unos usuarios desde un servicio. Esos usuarios los almacena en la propiedad "users". Además tenemos otra propiedad de tipo cadena que se llama "query".

```

import { Component, OnInit } from '@angular/core';
import { UsersService } from '../users.service';

@Component({
  selector: 'app-user-list',
  templateUrl: './user-list.component.html',
  styleUrls: ['./user-list.component.css']
})
export class UserListComponent implements OnInit {
  users = [];
  query = "";

  constructor(private userService: UsersService) {}

  ngOnInit(): void {
    this.userService.getUsers().subscribe(
      users => this.users = users
    )
  }
}

```

El template del componente hace uso del pipe que acabamos de crear, pero además tiene un campo de texto donde se puede escribir algo para realizar el filtrado. El campo de texto está asociado con la propiedad query del componente, de modo que cada vez que se escriba algo nuevo, se realice el filtrado entre los usuarios recibidos.

```

<div>
  Escribe el filtrado de usuarios: <input type="text" [(ngModel)]="query">
</div>

<pre>{{(users | nombre:query | json)}}</pre>

```

Fíjate que el listado de usuarios se muestra finalmente en el template como un JSON. Pero primero se aplica el filtrado por el nombre "nombre:query" y luego el pipe para convertir el array en un JSON. Como puedes ver, es perfectamente posible encadenar varios pipes de Angular si fuera necesario para conseguir varias transformaciones.

Conclusión

Los pipes son una herramienta indispensable para sacar todo el poder de Angular y conseguir transformaciones de los datos a la hora de mostrarlos en los templates. Usar pipes es inmediato, pero crearlos también resulta bastante sencillo.

En este artículo hemos podido hacer varios ejemplos con pipes. Si lo deseas puedes ver el código completo de la aplicación Angular de ejemplo en GitHub en este enlace:
<https://github.com/deswebcom/pipes-en-angular>

Este artículo es obra de *Miguel Angel Alvarez*
Fue publicado / actualizado en 16/04/2021
Disponible online en <https://desarrolloweb.com/articulos/crear-usar-pipes-angular>

Sistema de routing

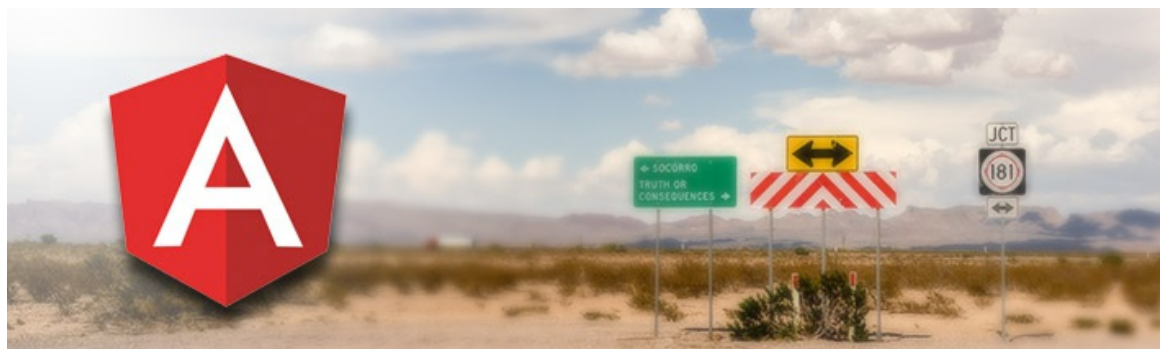
El sistema de routing es algo fundamental en las aplicaciones desarrolladas con Angular. En los siguientes artículos aprenderás a implementar rutas en aplicaciones web del estilo SPA (Single Page Application) y gestionar las rutas, de modo que se puedan pasar parámetros entre ellas, que los navegadores sean capaces de mostrar la ruta activa, etc.

Introducción al sistema de Routing en Angular

Qué es el sistema de routing, qué elementos contiene y cómo configurar las primeras rutas en una aplicación Angular.

En este artículo vamos a comenzar con una serie de entregas del Manual de Angular en las que vamos a abordar el sistema de routing, uno de los actores principales de las aplicaciones Angular.

El sistema de routing no es un asunto trivial, ya que envuelve a muchas clases, objetos, componentes y configuraciones. De hecho es algo bastante sofisticado, ya que tiene decenas de configuraciones para realizar rutas de todo tipo. No obstante, comenzar a trabajar con él en un primer ejemplo no es tan complicado, como podrás ver a continuación.



Qué es un sistema de routing

Pensando en aquellas personas que comienzan con Angular su incursión en las "SPA" (Single Page Application), comenzaremos aclarando qué es un sistema de routing y por qué lo necesitamos.

En cualquier sitio web generalmente tienes varias direcciones que son entregadas por un servidor, para mostrar diferentes contenidos del sitio. Podemos tener una portada, una página de productos, una de contacto, etc. Cada una de esas páginas se presenta en una ruta diferente del sitio web, que podrían ser como `example.com`, `example.com/productos/index.html`, `example.com/contacto.html`, etc. Cada una de esas rutas podría tener un archivo HTML, que se sirve con el contenido de esa sección. Hasta aquí estamos seguros que los lectores no

tendrán ningún problema, pues es así como funcionan, en líneas generales, prácticamente todos los sitios web.

Sin embargo, en las aplicaciones Angular sólo tenemos una página, el `index.html` y toda la acción se desarrolla dentro de esa página. En Angular lo común es que el `index` sólo tenga un componente en su `BODY` y realmente toda la acción se desarrollará en ese componente. Todas las "páginas" (pantallas o vistas) del sitio web se mostrarán sobre ese `index`, intercambiando el componente que se esté visualizando en cada momento.

Nota: Ese es básicamente el concepto de Single Page Application, del que puedes obtener más información en un artículo genérico sobre SPA, que explica con mayor detalle este modo común de desarrollo de aplicaciones con Javascript.

Para facilitar la navegación por un sitio donde realmente sólo hay un `index`, existe lo que llamamos el sistema de routing, que tiene el objetivo de permitir que en el sitio web haya rutas internas, respondiendo a rutas "virtuales" como las que existen en los sitios tradicionales.

Llamamos "virtuales" a esas rutas, porque realmente sólo existe un `"index.html"`, no habrá un archivo `"contacto.html"` o `"productos.html"` para cada ruta, sino que será realmente siempre el `"index.html"` el que se entregue al navegador.

El sistema de routing es el encargado de reconocer cuál es la ruta que el usuario quiere mostrar, presentando la pantalla correcta en cada momento. Esto es útil por varios motivos, entre ellos:

1. Permite que la aplicación responda a rutas internas. Es decir, no hace falta entrar siempre en la pantalla principal de la aplicación y navegar hasta la pantalla que queremos ver realmente.
2. Permite que el usuario pueda usar el historial de navegación, yendo hacia atrás y hacia adelante con el navegador para volver a una de las pantallas de aplicación que estaba viendo antes.

El sistema de routing de Angular

Angular, como un buen framework, dispone de un potente sistema de routing para facilitar toda la operativa de las single page applications. Está compuesto por varios actores que tienen que trabajar juntos para conseguir los objetivos planteados.

Comenzaremos explicando el sistema de routing resumiendo los elementos básicos que forman parte de él y que son necesarios para comenzar a trabajar.

- El módulo del sistema de rutas: llamado `RouterModule`.
- Rutas de la aplicación: es un array con un listado de rutas que nuestra aplicación soportará.
- Enlaces de navegación: son enlaces HTML en los que incluiremos una directiva para indicar que deben funcionar usando el sistema de routing.

- Contenedor: donde colocar las pantallas de cada ruta. Cada pantalla será representada por un componente.

Crear nuestra primera aplicación con rutas

Ahora que tenemos claro qué es el sistema de routing y sabemos qué elementos principales vamos a necesitar para ponerlo en marcha, vamos a crear una especie de receta para poder crear una aplicación, desde cero, en la que podamos incorporar rutas sencillas.

Nota: para comenzar esta práctica damos por hecho que tienes una aplicación vacía, en la que solo tenemos el módulo principal (app.module.ts) y el componente principal "app.component.js".

1.- Importar el código del sistema de routing

En nuestro módulo principal tenemos que hacer los correspondientes imports.

Deberías incluir a RouterModule y Routes, que obtenemos de la librería "@angular/router". El código será como este:

```
import { Routes, RouterModule } from '@angular/router';
```

RouterModule es el módulo donde está el sistema de rutas, por tanto contiene el código del propio sistema de rutas.

Routes es una declaración de un de tipo, que corresponde con un array de objetos Route. Los objetos Route están declarados por medio de una interfaz en el sistema de routing. Esta interfaz sirve para que en la declaración de tus rutas coloques solamente aquellos valores que realmente son posibles de colocar. Si no lo haces así, el compilador de TypeScript te ayudará mostrando los correspondientes errores en tiempo de desarrollo.

Nota: Deberías estar esperando que metamos el RouterModule en el imports del decorador del módulo principal, pero para hacer esto aún tenemos pendiente un paso intermedio.

2.- Crear la lista de las rutas disponibles en la aplicación

En este paso tenemos que crear un array de las rutas que queremos generar en nuestra aplicación. Utilizaremos la declaración del tipo Routes, que hemos dicho es un array de objetos Route.

Cada objeto Route del array tiene un conjunto de campos para definir la ruta, definido por la

interfaz Route, siendo que lo general es que tenga al menos un camino "path" y un componente para representar en esa ruta.

El código que producirás será más o menos así.

```
const rutas: Routes = [  
  { path: '', component: HomeComponent },  
  { path: 'contacto', component: ContactoComponent }  
];
```

Ese código también lo puedes colocar en el módulo principal, aunque nada te impide organizarlo mejor y colocarlo en un archivo aparte, exportando la constante "rutas" (el array) y luego importando desde el módulo principal.

Como puedes ver, los caminos (propiedad path) se definen con cadenas. La cadena vacía "" es el home de nuestra aplicación (URL como example.com). La cadena "contacto" corresponde a la ruta "/contacto" (URL como example.com/contacto).

Luego se indica el componente que debe mostrarse para cada ruta (propiedad component). Esos componentes podemos considerarlos como las pantallas de la aplicación. Tendremos uno por ruta, de modo que, cuando se acceda a tal ruta, Angular mostrará tal componente, y quitará el componente anterior que se estuviera mostrando.

Nota: obviamente ese componente debe formar parte del módulo o de otro módulo importado, es decir, el componente que representa la ruta debe ser conocido para poder usarse, como cualquier otro componente en general. Como ya sabes crear componentes creemos que no necesitarás muchas explicaciones. Pero generar esos componentes es tan sencillo como lanzar los comandos del CLI: "ng g c home" y "ng g c contacto". Si todo ha ido bien, esos componentes se habrán creado en el módulo principal y se habrán agregado en los imports del módulo y en el decorador, en el array "declarations". Por supuesto, puedes crear esos componentes en el módulo que mejor te venga.

3.- Declarar el sistema de routing en el "imports" del decorador @NgModule

Ahora tienes que editar el decorador del módulo principal @NgModule, donde tendrás que colocar en el array "imports" la declaración de que vas a usar el sistema de routing de Angular.

Tenemos que indicar a RouterModule, pero en ese imports necesita la configuración de rutas, creada en el array Routes[] del paso anterior. De momento, esa configuración la cargamos mediante un método forRoot() en el que pasamos por parámetro el array de rutas.

Se ve mejor con el código. Este es el decorador completo, pero te tienes que fijar especialmente en el array "imports".

```
@NgModule({  
  declarations: [  
    AppComponent,  
    HomeComponent,
```

```
ContactoComponent
},
imports: [
  BrowserModule,
  RouterModule.forRoot(rutas)
],
providers: [],
bootstrap: [AppComponent]
})
```

Con esto hemos terminado el trabajo en el módulo principal. Pero aun nos queda implementar la vista en el componente principal para generar los enlaces a las rutas y el contenedor donde se tienen que mostrar los componentes.

4.- Definir el template del componente principal

Comenzamos el trabajo en el componente principal. En este componente tenemos que meter un poco de código en la vista, archivo "app.component.html".

En el template se crea el navegador de las rutas, indicando en los enlaces la directiva "routerLink", con el valor de cada ruta.

```
<nav>
  <a routerLink="/">Home</a> |
  <a routerLink="/contacto">Contacto</a> |
</nav>
```

No puede haber ningún enlace generado, que no tenga su correspondencia en el sistema de rutas. Fíjate que los enlaces comienzan por una "/" para hacerlos absolutos a la raíz del dominio.

Por último debes colocar un contenedor llamado "router-outlet" para definir el espacio donde se van a inyectar los componentes. Es tan sencillo como colocar esta etiqueta definida por el módulo del sistema de routing.

```
<router-outlet></router-outlet>
```

Nota: Aunque parezca un componente, "router-outlet" es una directiva, ya que su función es manipular el DOM para inyectar un componente u otro frente al cambio de una ruta.

Con eso es todo. Ahora si abres tu aplicación deberías ver las rutas funcionando.

Problemas comunes con el sistema del routing

Lo que hemos hecho hasta ahora es poco, por lo que no debería haber muchos problemas con nuestro ejemplo, pero vamos a ver un par de posibles errores que nos podemos encontrar.

Si nos equivocamos al escribir en un enlace la ruta, obtendremos un mensaje de error al hacer clic sobre ese enlace.

Por ejemplo escribimos "contato" en vez de "contacto" en la ruta.

```
<a routerLink="/contato">Contacto</a> |
```

Este es el error que vamos a encontrar: `_Error: Cannot match any routes. URL Segment: 'contato' _`

Hasta ahora las rutas estaban asociadas a un componente. Puede ocurrir que en la declaración del array de Routes indiques un componente que no existe, ya sea porque has escrito mal el nombre de la clase del componente, o porque estás intentando usar un componente que no está en este módulo o que no está importado por este módulo.

Por ejemplo, podemos colocar el nombre de un componente que no existe como ruta:

```
{ path: "", component: HComponent },
```

En este caso el editor probablemente te alertará del error, señalando con una línea en rojo, pero si no lo ves, el compilador de TypeScript será el que te avise. El mensaje de error que recibiremos será algo como: `Cannot find name 'HComponent'`.

Eso era porque HComponent no existía, pero podría ocurrir que el componente sí que exista, lo hayas importado, pero no esté disponible en el módulo principal. Entonces el error sería algo como `_"CompNoImportadoComponent is not part of any NgModule or the module has not been imported into your module" _`.

Otra cosa que te puede ocurrir es que coloques una "/" al principio de una ruta. Fíjate que en la configuración de las rutas la ruta raíz es "" (cadena vacía) y no "/".

Esto sería incorrecto:

```
const rutas: Routes = [  
  { path: '/', component: HomeComponent },  
  { path: '/contacto', component: ContactoComponent }  
];
```

Para advertirte que las rutas no comienzan por "/", Angular te mostrará el siguiente error en la consola del navegador: `_"Invalid configuration of route '/': path cannot start with a slash" _`.

La verdad es que el editor te ayuda mucho para no cometer errores, gracias al soporte de TypeScript, además que los errores de Angular son bastante claros. Si no encuentras aquí tu problema y no eres capaz de entenderlo, siempre puedes googlear ;).

Conclusión

Esta introducción al sistema de routing de Angular ha abordado tanto la teoría relacionada con la necesidad de uso de un sistema de routing, como la práctica en su implementación en Angular.

No obstante, como ya hemos comentado, el sistema de routing es muy complejo y tiene muchas otras cosas que debemos aprender para sacarle todo el partido y cubrir necesidades comunes de las aplicaciones. Seguiremos abordando las rutas y su configuración en los próximos artículos del manual.

Este artículo es obra de *Miguel Angel Alvarez*
Fue publicado / actualizado en 30/04/2018
Disponible online en <https://desarrolloweb.com/articulos/introduccion-sistema-routing-angular.html>

Rutas en Angular: mostrar la ruta activa

Cómo mostrar la ruta activa en el navegador de una aplicación Angular, aquella que se está mostrando en este momento.

Después de incursionar en el sistema de routing de Angular, en un artículo que contuvo mucha información, vamos a realizar un paso bastante más simple, pero esencial para la usabilidad de las aplicaciones. Se trata de marcar la ruta activa en el navegador.

Por supuesto, la aplicación de estilos es una responsabilidad del CSS, por lo que tendremos una clase "activo" o con algún nombre similar, que usaremos para aplicar el estilo. Sin embargo esa clase se debe aplicar dinámicamente al elemento que toque en cada momento y eso es lo que tenemos que hacer con Angular.

Afortunadamente, esta parte es bastante sencilla, gracias a una directiva que permite aplicar una clase CSS cuando el navegador está mostrando una ruta determinada.



Directiva routerLinkActive

La directiva de Angular routerLinkActive es la que nos permite aplicar una clase dinámicamente, dependiendo de si un enlace determinado tiene su ruta activa. La forma de trabajar en principio es bien sencilla, como veremos a continuación.

Como cualquier cosa en Angular, comenzamos importando en el TypeScript del componente la directiva que pretendemos usar.

```
import { RouterLinkActive } from '@angular/router';
```

En el siguiente código tenemos una primera aproximación, aplicando la directiva routerLinkActive al navegador, en nuestro template. Aunque todavía no es exactamente la más completa, enseguida veremos por qué.

```
<nav>
  <a routerLink="/" routerLinkActive="activo">Home</a> |
  <a routerLink="/contacto" routerLinkActive="activo">Contacto</a> |
  <a routerLink="/quienessomos" routerLinkActive="activo">Quienes somos</a>
</nav>
```

Como puedes comprobar en el código anterior, se ha aplicado una clase llamada "activo", por lo que tendrás que crear esta clase en el CSS del componente, claro está. Si no, no habrá ningún CSS que aplicar.

Un detalle interesante con respecto a Angular es que hemos colocado un string literal como valor a la directiva. Por eso la directiva no se aplica con los corchetes, porque lo que hay entre comillas como valor no es una variable o una expresión que se tenga que evaluar, sino simplemente una cadena literal: "activo".

Nota: Observa que hemos creado una ruta adicional, con respecto al ejercicio tal como lo dejamos en el artículo anterior. Esto te obligará a practicar un poco, creando la ruta en el array de Routes y luego el componente correspondiente. Esto te lo dejamos para ti.

Aplicación de routerLinkActive sobre el contenedor

Hay veces que el CSS no lo quieres aplicar al propio enlace, sino a su contenedor. Esto no es un problema para routerLinkActive, que también funciona si tienes que aplicar esa clase dinámicamente al padre donde está el enlace de navegación.

Por ejemplo imagina que tienes una lista y que el estilo del elemento activo debe colocarse en el elemento LI. No habría problema para conseguir ese detalle, de la siguiente manera.

Nuestro template podría entonces tener esta forma:

```
<nav>
<ul>
<li routerLinkActive="activo">
<a routerLink="/">Home</a>
</li>
<li routerLinkActive="activo">
<a routerLink="/contacto">Contacto</a>
</li>
<li routerLinkActive="activo">
<a routerLink="/quienessomos">Quienes somos</a>
</li>
</ul>
</nav>
```

Y aunque este no es un Manual de CSS, podríamos aplicar unos estilos como estos:

```
.activo {
background-color: #ccc;
}
.activo a {
color: green;
}
ul {
list-style: none;
}
li {
display: inline-block;
margin-right: 20px;
background-color: #eee;
padding: 15px;
}
```

Aplicar estilo sólo a la ruta exacta

Si ejecutas tu aplicación, tal como la hemos dejado, podrás apreciar que tenemos un pequeño problema. El enlace de "home" aparece siempre como activo. Es decir, navegamos a las otras secciones del sitio, que se marcan correctamente como activo, pero el enlace de "home" siempre permanece como activo también.

No es un error, está funcionando tal como se espera. Por defecto, routerLinkActive marca como activo a un enlace del navegador que contenga la ruta, no que sea exactamente esa ruta.

Me explico. En rutas como <http://localhost:4200/quienessomos> la ruta exacta es "/quienessomos", pero la ruta raíz "/" está contenida en "/quienessomos". Por tanto ocurre que al visitar "/quienessomos" se marcan como activos dos enlaces, el raíz y el "quienessomos".

Esto podría no ser un problema en algunos casos, pero si no te gusta se puede solucionar perfectamente. Para ello tenemos que conocer un configurador llamado "routerLinkActiveOptions".

Nota: Como enseguida verás en el código, routerLinkActiveOptions podría parecer una nueva directiva, pero realmente no lo es. Forma parte de la directiva routerLinkActive y sirve para configurar el comportamiento de ésta. Como no es una directiva, sino que forma

parte de la directiva que ya venimos usando, no hace falta que hagamos el import de `routerLinkActiveOptions`.

Esta configuración de la directiva nos aporta la posibilidad de configurar, mediante un objeto, algunas opciones para marcar el enlace activo. La que necesitamos usar se llama "exact". Aplicada a nuestro template, nos quedaría más o menos como esto:

```
<nav>
<ul>
<li routerLinkActive="activo" [routerLinkActiveOptions]="{exact: true}">
  <a routerLink="/">Home</a>
</li>
<li routerLinkActive="activo">
  <a routerLink="/contacto">Contacto</a>
</li>
<li routerLinkActive="activo">
  <a routerLink="/quienessomos">Quienes somos</a>
</li>
</ul>
</nav>
```

Fíjate en los siguientes puntos:

- Ahora sí necesitamos poner usar la notación de corchetes en `routerLinkActiveOptions`, porque lo que estamos pasando no es una cadena, sino algo de Javascript que se tiene que evaluar. En este caso le pasamos un objeto.
- En el objeto, colocamos "exact" con el valor true, lo que indica que este enlace sólo se marcará como activo si la ruta exacta es "/".

Ahora, si ejecutas tu ejemplo podrás observar que está tal como hubiéramos deseado, marcando sólo como activa la ruta que se necesita en cada caso.

Conclusión

Ya tenemos las herramientas necesarias para hacer un bonito navegador, usando rutas internas de la aplicación. Nuestro sistema de routing en Angular está cada vez más configurado y gracias a este conocimiento podremos hacer aplicaciones con un aspecto más profesional.

Este artículo es obra de *Miguel Angel Alvarez*
Fue publicado / actualizado en 15/05/2018
Disponible online en <https://desarrolloweb.com/articulos/rutas-angular-mostrar-activa.html>

Parámetros en las rutas Angular

Cómo trabajar con parámetros en las rutas, configurar el sistema de routing para crear rutas que envían parámetros y recibir los parámetros en los componentes.

En los pasados artículos hemos podido aprender a [trabajar con el sistema de routing de Angular](#), pero existe una situación muy típica que no hemos tenido tiempo de conocer todavía y que vamos a abordar ahora. Se trata de generar rutas que aceptan parámetros.

En principio puede ser una tarea sencilla, pero hay varios asuntos delicados que se deben conocer para hacer las cosas bien. No se trata solamente de crear enlaces que envían parámetros, sino aprender a recibir esos parámetros. Además, cuando los parámetros cambian, saber detectar esos cambios de manera asíncrona. Vayamos poco a poco para que sea fácil de entender.



Declarar rutas que utilizan parámetros

El primer paso será declarar en nuestro listado de rutas la posibilidad de recibir parámetros. Para ello usaremos una notación especial que te resultará familiar si has usado otros sistemas de routing en el pasado. Podemos verla a continuación.

```
{ path: 'camino/:parametro', component: ParametroComponent }
```

Nota: ten en cuenta que esta declaración de ruta es un elemento del array `Routes`. Lee el artículo de introducción al sistema de routing para más información. Luego en este artículo encontrarás un ejemplo completo de declaración de todas las rutas en el array.

En resumen, al crear el camino, estamos indicando varios segmentos, separados por `"/"`. En alguno de esos segmentos estamos colocando el carácter `":"` delante, con el que se indica que eso no es una cadena estática, sino un parámetro.

La diferencia entre `"camino"` y `":parametro"` es que la ruta comenzará siempre por `"camino"`, la cadena literal. Sin embargo, donde colocamos `":parametro"` quiere decir que nos acepta cualquier texto, es decir, no es una cadena literal específica, sino cualquier cadena que se pueda recibir.

Por tanto, esa ruta parametrizada será siempre servida por el componente `"ParametroComponent"`, y aceptará rutas diversas, como podría ser:

- `example.com/camino/1`

- `example.com/camino/2`
- `example.com/camino/otro`
- `example.com/camino/cualquier_cosa`

Podemos indicar cualquier número de parámetros en una declaración de ruta. En ese caso usamos la notación de los ":" por cada parámetro que se quiera definir. Por ejemplo podríamos tener algo como esto:

```
{ path: 'coches/:marca/:modelo', component: CochesComponent }
```

En la ruta declarada tenemos dos parámetros "marca" y "modelo". Nos aceptará rutas como las siguientes:

- `example.com/coches/seat/ibiza`
- `example.com/coches/fiat/panda`
- `example.com/coches/audi/a8`

El número de parámetros no acepta ni más ni menos de los indicados

Pero, ojo, para que la ruta concuerde con la anterior declaración, el número de parámetros definidos en las rutas debe ser exacto. Es decir, la declaración anterior no aceptará rutas como estas:

- `example.com/coches`
- `example.com/coches/ford`

Si queremos atender a estas rutas, deberíamos crear una nueva declaración en el array Routes, para la ruta que acepte solo el segmento "coches" y el segmento con un único parámetro "coches/:marca".

Generar enlaces a rutas enviando parámetros

El siguiente paso que tendremos que hacer es generar enlaces a esa ruta, enviando parámetros. Esta parte no tiene ningún secreto en especial, simplemente tendremos la ruta en la que enviaremos los valores que se deseen como parámetros.

```
<a routerLink="/coches/seat/ibiza">Seat Ibiza</a>
```

Como puedes ver, esa ruta casa con nuestra declaración, enviando una marca y un modelo de coche determinado.

Además, podemos crear las rutas mediante la directiva "routerLink" con una notación de array, lo que tendrá mucho sentido más adelante, cuando partes de esa ruta sean dinámicas y correspondan con una propiedad del componente.

```
<a [routerLink]="['/coches', 'ford', 'focus']">Ford Focus</a>
```

En este caso, por pasar un array como valor a la directiva routerLink, estamos obligados a colocar el nombre de la directiva entre corchetes, ya que el valor asignado no es un simple literal de cadena. Como puedes ver, cada segmento de la ruta corresponde con una casilla del array.

En este ejemplo quizás no tenga mucho sentido usar esa notación de array, pero como decía, si la marca o modelo fueran propiedades del componente, sí que sería muy útil.

```
<a [routerLink]="['/coches', miMarca, miModelo]'>Ford Focus</a>
```

En este ejemplo "miMarca" y "miModelo" se supone serían variables dinámicas, por lo tanto este enlace nos llevará a un sitio u otro dependiendo de esas propiedades del componente.

Recibir valores de los parámetros

Ahora viene la parte más interesante, en la que vamos a recuperar los parámetros enviados mediante el sistema de rutas. Esta parte se puede complicar un poco, como luego veremos. Para hacerlo más sencillo aplicaremos distintas alternativas, con dificultad creciente.

En cualquiera de las alternativas, para recibir los valores de los parámetros en el componente al que nos dirige la ruta tenemos que usar un objeto del sistema de routing llamado "ActivatedRoute". Este objeto nos ofrece diversos detalles sobre la ruta actual, entre ellos los parámetros que contiene.

Para usar este objeto tenemos que importarlo primero. Lo haremos en el componente donde vas a recibir los parámetros, claro.

```
import { ActivatedRoute, Params } from '@angular/router';
```

Luego tendrás que inyectar ese objeto en el constructor.

```
constructor(private rutaActiva: ActivatedRoute) { }
```

Una vez inyectado ya lo puedes usar dentro del componente, para obtener los datos que necesitas. Dada la declaración anterior, la propiedad del componente "rutaActiva" será la que contenga el objeto que vamos a necesitar para recuperar las rutas.

Recibir valores de parámetros con el snapshot

La manera más sencilla de comenzar es usar el snapshot de los valores de los parámetros. el problema de esta alternativa es que no siempre funcionará, pero vamos a verla primero porque nos resultará más fácil de entender.

Una vez inyectada la ruta activa mediante el objeto de tipo ActivatedRoute, podemos acceder a

los parámetros en el método `ngOnInit()`.

El objeto `ActivatedRoute` lo hemos recibido por medio de una propiedad del componente llamada `rutaActiva`. Para acceder a un snapshot de los parámetros en un instante dado usamos este código:

```
this.rutaActiva.snapshot.params
```

El snapshot te da los parámetros del componente en el instante que los consultes. El objeto `params` contendrá todas las propiedades según los parámetros recibidos. Quiere decir que, el modelo lo recuperaremos con `"this.rutaActiva.snapshot.params.modelo"` y la marca con `"this.rutaActiva.snapshot.params.marca"`.

Por tanto, este podría ser el código de nuestro componente `"CochesComponent"` hasta ahora:

```
import { Component, OnInit } from '@angular/core';
import { ActivatedRoute } from '@angular/router';

@Component({
  selector: 'app-coches',
  templateUrl: './coches.component.html',
  styleUrls: ['./coches.component.css']
})
export class CochesComponent implements OnInit {
  coche: {marca: string, modelo: string};

  constructor(private rutaActiva: ActivatedRoute) {}

  ngOnInit() {
    this.coche = {
      marca: this.rutaActiva.snapshot.params.marca,
      modelo: this.rutaActiva.snapshot.params.modelo
    };
  }
}
```

Fíjate en los siguientes puntos:

- El import de `ActivatedRoute`
- La declaración de la propiedad `coche` del componente. Esta declaración ya viene con un tipo declarado, creado por medio de un objeto "inline". Esto es como si hubiésemos hecho una interfaz y hubiésemos dicho que el objeto "coche" es de esa interfaz. Pero en lugar de ello estoy declarando la interfaz en la misma declaración de la propiedad.

Nota: Quizás esa declaración de una interfaz "en línea" es un poco compleja de ver y puedas pensar que es innecesario. De hecho, no se necesita realmente, sólo lo he hecho para crear un tipo concreto a la propiedad "coche", pues si no lo hago el editor se queja y me muestra errores en tiempo de desarrollo que me molestan a la hora de programar. Lo ideal sería definir el modelo de datos, para el tipo coche en un archivo aparte e importarlo en el componente.

- La inyección del ActivatedRoute en el constructor
- La inicialización de la propiedad "coche" dentro de ngOnInit()
- En esa inicialización es donde accedemos al snapshot y colocamos cada uno de los parámetros recibidos como propiedades del objeto coche.

Ahora, en nuestro template, podremos usar la propiedad coche, inicializada gracias al ngOnInit() y el acceso a los parámetros recibidos.

```
<h1>
  Coches en venta
</h1>
<p>
  Este es un coche marca: {{coche.marca}}
</p>
<p>
  Modelo: {{coche.modelo}}
</p>
```

Te preguntarás ¿Por qué esta aproximación de la recepción de parámetros en Angular podría no funcionar? El caso es que nuestro componente podría no ser consciente de que los parámetros han cambiado en un momento dado.

Tal como hemos dejado el ejemplo hasta el momento, si has ido programando en tu propio proyecto con estas indicaciones, podrás encontrar el problema, si cambias de una ruta a otra dentro del mismo componente. La primera te recoge bien los parámetros, pero si ya estás en "CochesComponent" y pulsas un enlace a la misma ruta pero con valores de parámetros distintos, observarás que no se refrescan los valores en el template.

El problema es que la inicialización de los parámetros la hemos colocado en el ngOnInit() y si el componente ya pasó su etapa de inicialización, no se vuelven a recibir los parámetros con el snapshot. Por eso, si cambian los parámetros estando ya en el componente "CochesComponent", no se reciben los nuevos valores.

Recibir valores de parámetros con un observable

Los observables de Angular y RxJS nos vienen a solucionar esta situación. Como sabes, los observables nos permiten suscribirnos a eventos que se recibirán de manera asíncrona, mediante programación reactiva y manteniendo un alto rendimiento.

Nota: si no conoces los observables tenemos una introducción que te los explica de manera bastante detallada. Lee el artículo [Introducción a observables en Angular](#).

Básicamente, en el objeto de tipo ActivatedRoute tenemos una propiedad llamada "params" que es un observable y que nos sirve para suscribirnos a cambios en los parámetros enviados al componente.

Las suscripciones a los cambios en los observadores se realizan mediante el método subscribe().

Ese método recibe varios parámetros y el que nos interesa de momento es solo el primero, que consiste en una función callback que se ejecutará con cada cambio de aquello que se está observando.

En resumen, "this.rutaActiva.params" es el observable y "this.rutaActiva.params.subscribe()" es el método para suscribirnos a los cambios. A suscribe le enviaremos una función callback, la cual recibirá el nuevo valor, y se ejecutará cada vez que cambie.

La función callback recibirá un objeto de clase Params. Ese objeto también lo vas a tener que importar desde "@angular/router".

```
import { ActivatedRoute, Params } from '@angular/router';
```

El código de la suscripción es el siguiente.

```
this.rutaActiva.params.subscribe(  
  (params: Params) => {  
    this.coche.modelo = params.modelo;  
    this.coche.marca = params.marca;  
  }  
);
```

Nota: usamos una función flecha para poder seguir usando la variable "this". Puedes aprender sobre esto en el artículo [Arrow Functions de ES6](#).

Veamos ahora cómo quedaría el código de nuestro componente. En este caso el se nos complica algo más, pero si ya estás familiarizado con los observables no te resultará en ningún problema.

```
import { Component, OnInit } from '@angular/core';  
import { ActivatedRoute, Params } from '@angular/router';
```

```
@Component({  
  selector: 'app-coches',  
  templateUrl: './coches.component.html',  
  styleUrls: ['./coches.component.css']  
})  
export class CochesComponent implements OnInit {  
  coche: {marca: string, modelo: string};  
  
  constructor(private rutaActiva: ActivatedRoute) {}  
  
  ngOnInit() {  
    this.coche = {  
      marca: this.rutaActiva.snapshot.params.marca,  
      modelo: this.rutaActiva.snapshot.params.modelo  
    };  
    this.rutaActiva.params.subscribe(  
      (params: Params) => {  
        this.coche.modelo = params.modelo;  
        this.coche.marca = params.marca;  
      }  
    );  
  }  
}
```

Con esta modificación podemos estar seguros que, si accedemos a nuevas URL que simplemente envían nuevos parámetros al componente, Angular actualizará el template, mostrando los nuevos datos recibidos.

si te sales del componente para ir a otras rutas se debería cancelar la suscripción en el método `ngOnDestroy()`. Sin embargo, esto es algo que Angular ya realiza por ti mismo en el caso de las suscripciones a cambios en parámetros de las rutas, por lo que para este ejemplo en concreto no te necesitas preocupar.

Conclusión

Con esto hemos terminado nuestras explicaciones sobre parámetros en las rutas, enviados a componentes de Angular. Hemos visto cómo declarar rutas que reciben parámetros, cómo generar enlaces que hacen envío de valores para esos parámetros y cómo recibirlos mediante dos aproximaciones distintas.

La aproximación de los snapshots puede ser suficiente si estás seguro que los datos de los parámetros no van a cambiar durante la vida del componente. La aproximación con observables es más fiable, porque funcionará siempre, cambien o no cambien los valores de los parámetros enviados. Por tanto, es una buena práctica usar observables y será absolutamente necesario si los parámetros en la ruta pueden cambiar durante la vida de un componente.

Este artículo es obra de *Miguel Angel Alvarez*
Fue publicado / actualizado en 05/06/2018
Disponible online en <https://desarrolloweb.com/articulos/parametros-rutas-angular.html>

Apéndice

Otros temas de interés relativos al desarrollo con el framework Javascript Angular para el desarrollo de aplicaciones web. Una serie de artículos de temas diversos que no siguen un orden especial y que pueden resultar de interés para los desarrolladores Angular o TypeScript.

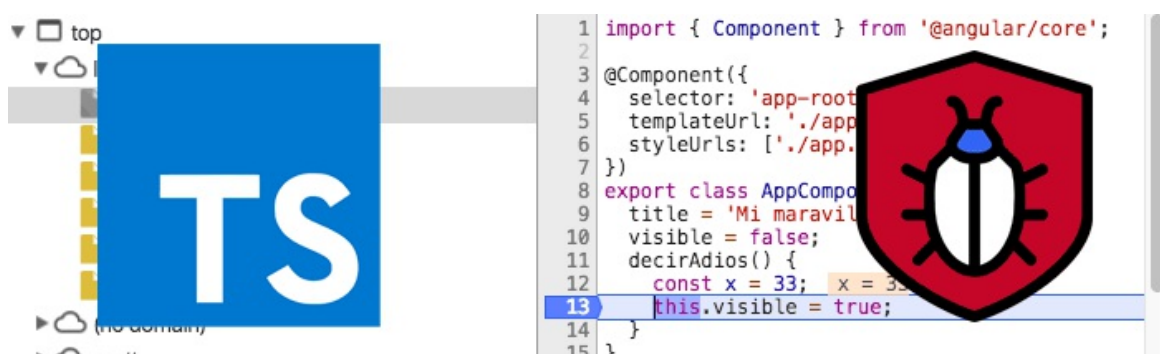
Cómo se depura el TypeScript en una aplicación Angular

Cómo depurar usando el navegador, por medio de las herramientas de debug del inspector, del código TypeScript que escribimos para las aplicaciones de Angular.

En este artículo del Manual de Angular no vamos a explicar cosas que dependen directamente del framework, sino que son más relacionadas con el lenguaje TypeScript, Webpack y el navegador que uses para la depuración de código.

Aunque se ha de admitir que no siempre es recomendable usar el debugger, porque es mucho más útil y ágil poner en marcha [técnicas de desarrollo de pruebas o testing en Angular](#), queremos mostrarte cómo usar el debugger con archivos ".ts", código TypeScript. Es útil en casos puntuales, cuando queramos usar el inspector del navegador y detener la ejecución del código en cierto punto, donde podamos examinar el estado de las variables.

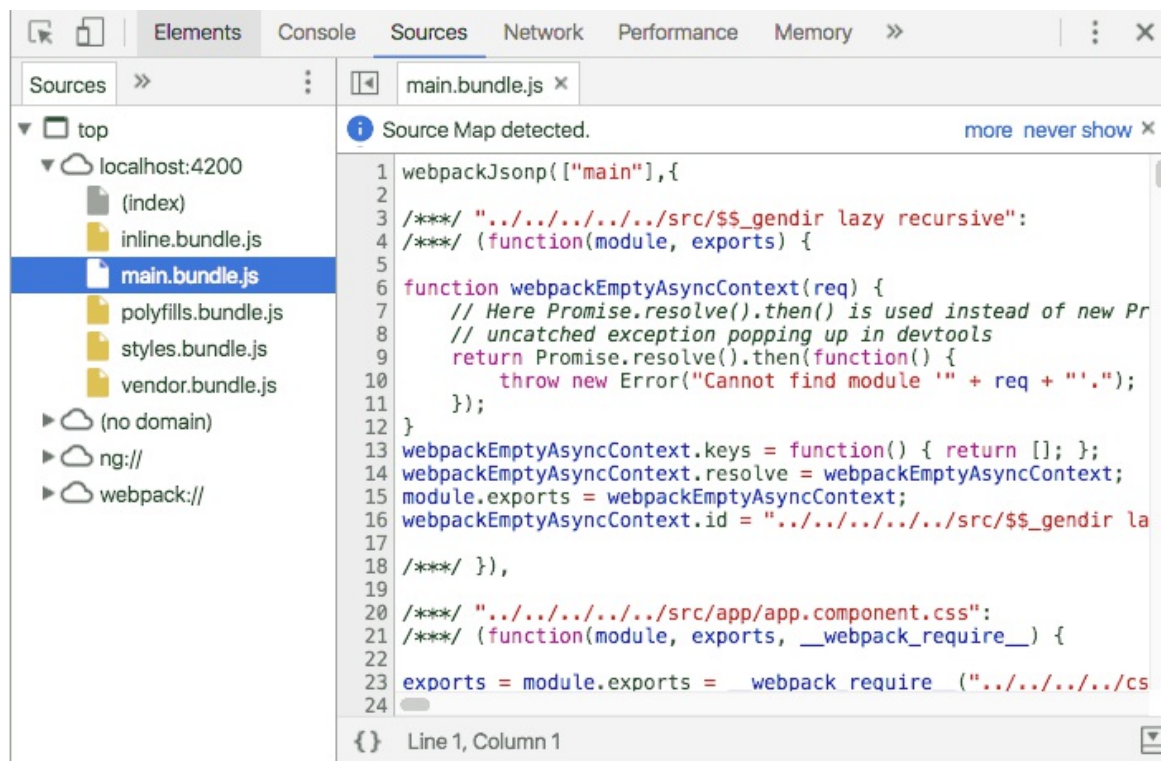
Esta tarea podría ser trivial, sin embargo, nuestro código es TypeScript y el navegador ejecuta Javascript. ¿Qué pasa en estos casos? Bien, pues en este artículo vamos a resolver una duda de la operativa de depuración de aplicaciones y el uso del debugger del navegador.



El problema del debug de TypeScript

En realidad el lenguaje TypeScript no tiene ningún problema con la depuración, gracias al trabajo realizado por debajo por Webpack. Pero si no se sabe cómo depurar, nos podemos ver en un problema bastante común, ya que tu código TypeScript se traduce a Javascript y cuando te dispones a analizar los sources en el navegador, encontrarás que el código que aparece no es el tuyo.

En esta imagen podemos ver la pestaña de sources de la consola. Observarás que los archivos de código no son los mismos que tienes a la hora de desarrollar en el navegador, ya que tenemos archivos como "main.bundle.js" que nosotros no hemos creado. Al observarlos, en el panel de la derecha, encontramos código Javascript que no es nuestro.



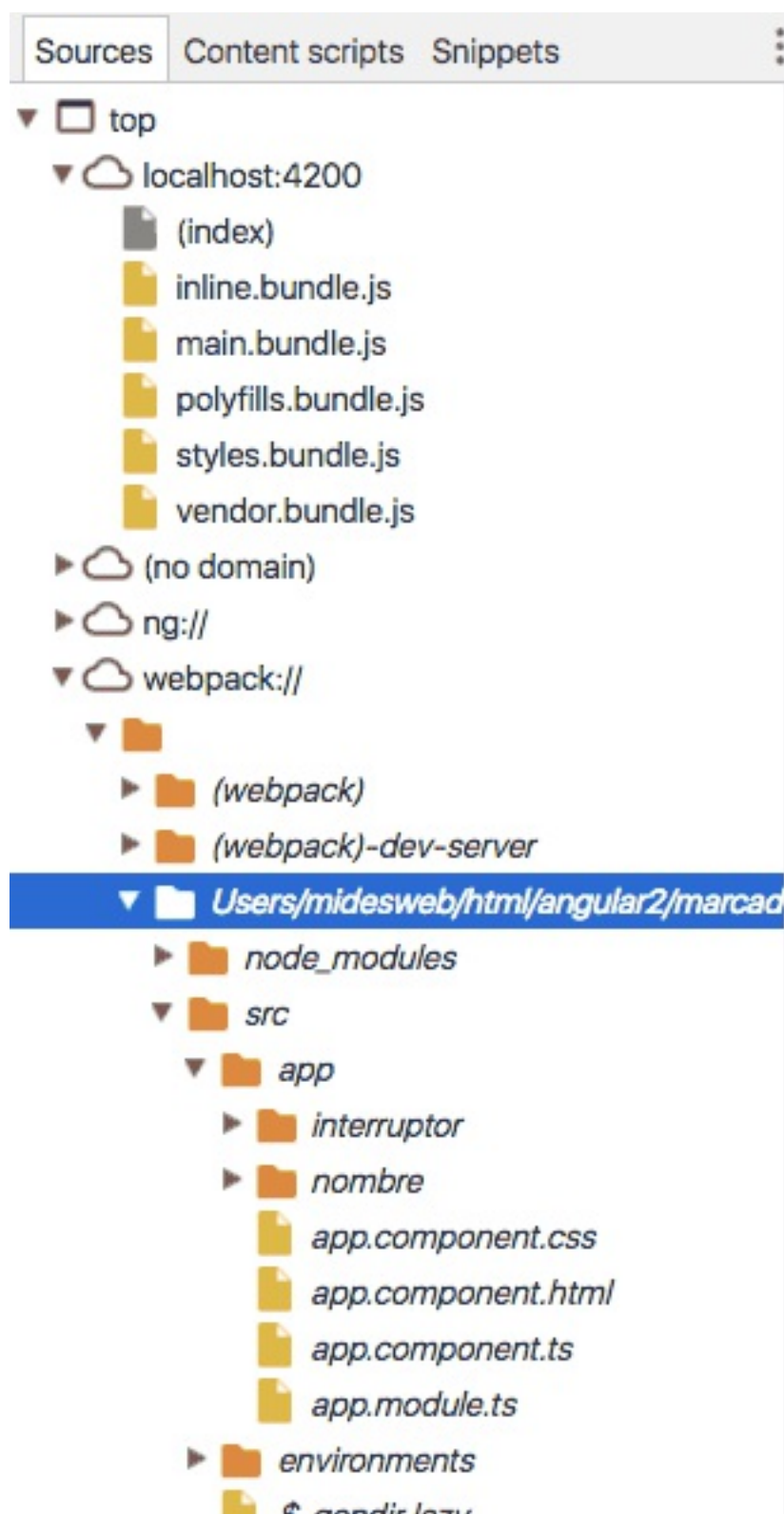
Nota: Webpack es el que se encarga de generar los bundles de código de la aplicación, ya sea al ver la app en el navegador en tiempo de desarrollo o al hacer el "ng build" para llevar a producción.

Depurar ese código sería posible, pero sirve de poco o nada, puesto que no lo hemos escrito nosotros, sino que ha sido generado. Tú generalmente querrás depurar en tu propio código, agregando los puntos de ruptura y examinando tus propias variables.

Dónde está nuestro código original

Nuestro código, tal como lo hemos escrito, lo podemos ver también por medio de la consola del navegador, aunque está un poco más escondido.

Lo podemos encontrar un poco más abajo en el listado de sources, en la sección "webpack://". Podemos desplegar las carpetas y encontraremos una en la que aparece la ruta de nuestro disco duro donde se encuentran los archivos de la aplicación. Está marcada en azul en la siguiente imagen.



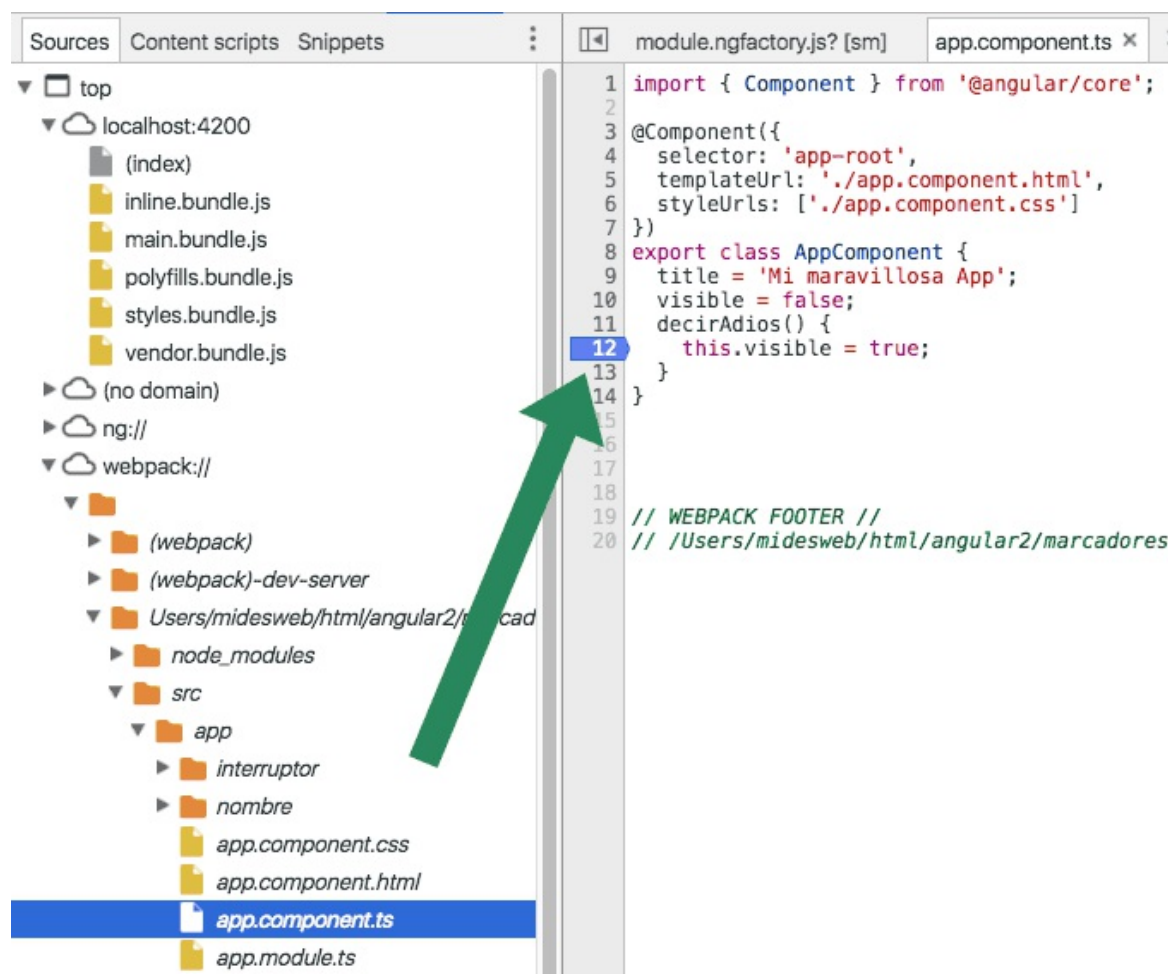
Allí podrás ver toda la estructura real de carpetas de tu app. Por tanto, encontrarás el directorio "src" y la carpeta "app", con todos los componentes y módulos que hayas ido creando. Lo más importante, encontrarás los archivos TypeScript originales (lo puedes ver por la extensión ".ts").

Nota: WebPack además ha hecho internamente un mapa, para corresponder el código de

esos scripts TypeScript con el código Javascript que realmente está ejecutándose en el navegador.

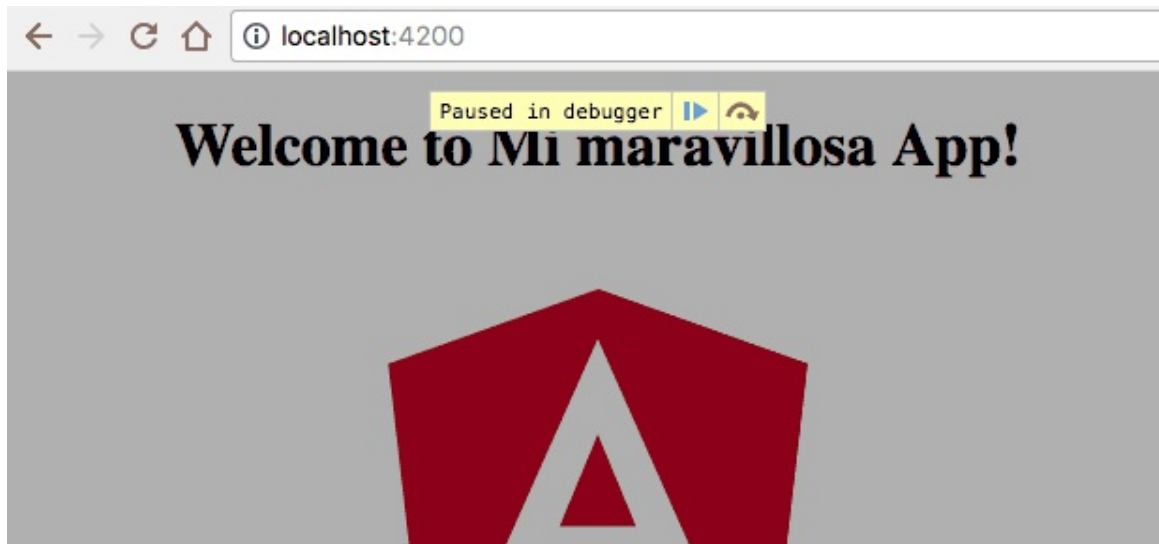
Poner puntos de ruptura

Ahora puedes poner puntos de ruptura del código, igual como si depurases una aplicación Javascript de toda la vida. Simplemente tocando el número de línea donde parar el flujo de ejecución, con un clic de ratón. Se marcará en azul como puedes ver en la siguiente imagen.

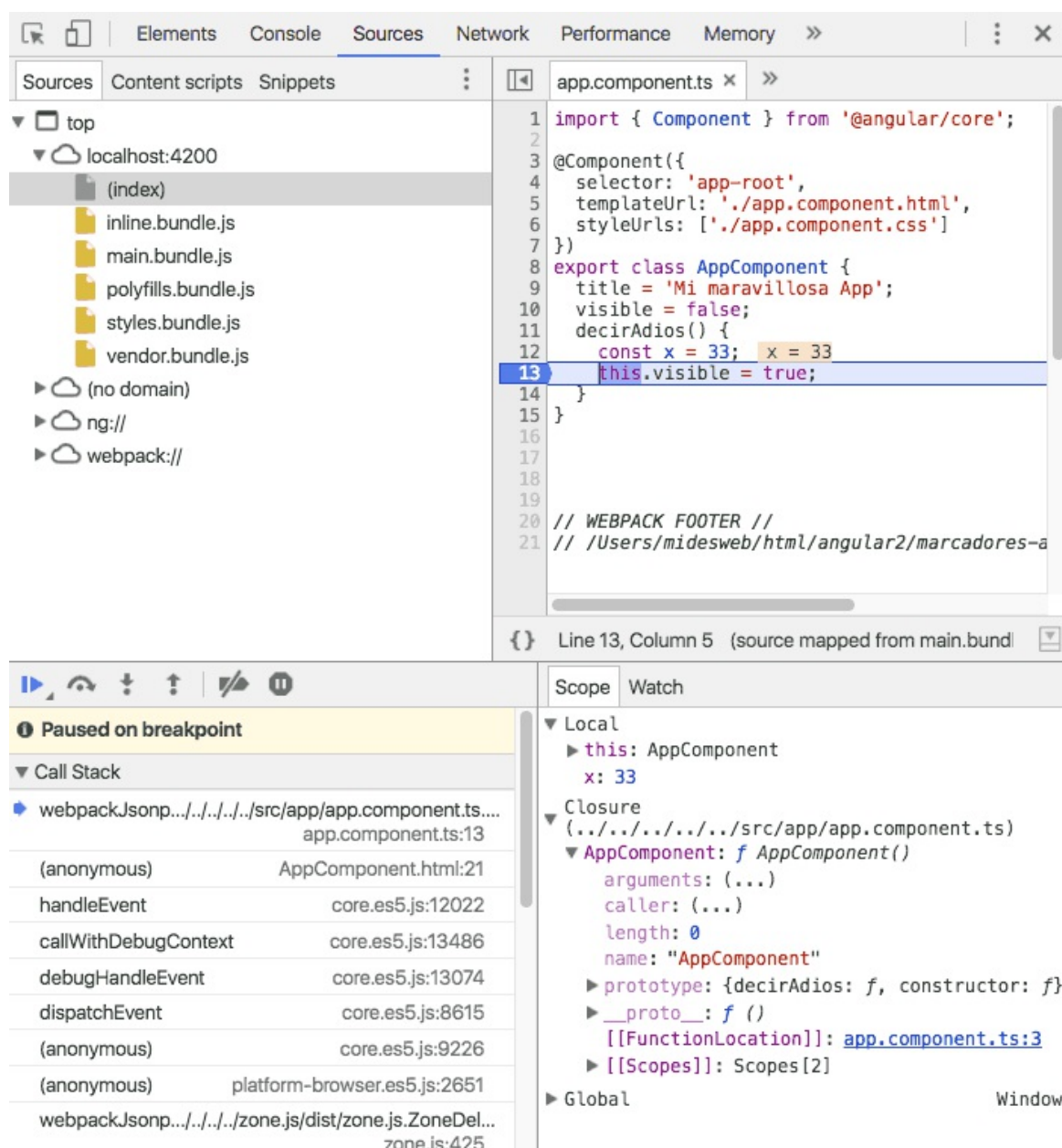


Ahora puedes recargar la página y el navegador debería detener la ejecución del código en ese punto, cuando pase por ahí el flujo de ejecución.

El navegador te informará que la ejecución se ha detenido por el debugger.



Y además podrás encontrar información en el panel de scope de los valores de las variables que existan en este momento, con sus valores.



Ya luego se trata de usar el debugger como ya debes de conocer, aunque a decir verdad, el hecho de que el código que se ejecuta no sea el tuyo seguirá interfiriendo en algunos casos, como cuando quieras meterte dentro de funciones para seguir la traza de ejecución.

De todos modos, con lo que hemos visto serás capaz de examinar un poco mejor tu código con el debugger, en busca de posibles problemas que de otro modo te resulte complicado de encontrar. Esperamos haberte ayudado así a depurar un poco mejor tus aplicaciones desarrolladas con Angular y TypeScript.

Este artículo es obra de *Miguel Angel Alvarez*
 Fue publicado / actualizado en 16/05/2018
 Disponible online en <https://desarrolloweb.com/articulos/depurar-typescript-angular.html>

